>> Welcome to ECE 530 Cloud Computing. My name is [inaudible]. Today I'm going to cover the introduction to cloud computing. The objective of this module overall is for students completing this module, they will be able to explain the motivation behind cloud computing, and how it is used nowadays. The development of file systems, and the history of cyberinfrastructure. What is the cloud? This is a common question that we hear nowadays. Cloud is a very important part of a number of businesses. Effectively there is no such thing as the cloud. It just probably someone else's computer. What does this mean? It means that we're using infrastructure, we're using machines, we're using computers, but they don't belong to us. But we're able to connect to them through the network and that has been possible because of the broadband adoption of broadband networks. The other question that we have to answer is whether cloud computing is important, and what we're trying to see in this table here, is the growth of the market of cloud computing. In this picture that the source is Gartner, that was the data collected in April 2017, about a year ago. What this indicates is the growth of cloud computing. Cloud expenses in four years or in three years from now, will actually almost double to $302 billion. We're talking about a huge business. While some of the definition here is like cloud business process services, cloud application infrastructure services, cloud application services, cloud management security services, and cloud system infrastructure services are not very clear to you right now, but these are exactly the terminology we're going to cover in this course. But the important aspect here is to understand that cloud computing is growing and it's going to grow a lot in the future. What we see as a distribution is most of actually the revenue will come from software as a service, that means software that will be built on top of the cloud to serve everyday needs. I can give you a few examples even from today. Today we see companies like Netflix, they call themselves cloud-native, that means they're under all infrastructure inside cloud computing. Many other common is Facebook, is introducing a cloud. Google has been using a fairly big data center deployment and now they move towards offering that as a public cloud offering which is named as Google Cloud. Amazon is probably the largest cloud vendor right now, which is used. How did it start? It started because they had so much computing resources that they were not using all they time. That means they had available resources to provide to other users. This is another study. The study comes from IDC 2016, about two years ago. This shows the worldwide spending on public cloud computing. That means that how many users pretty much and how much you're going to spend once they start on boarding cloud computing. We see the goals going to be tremendous. From 2018-2020, we're seeing about $50 billion will be spent on that and the growth will not stop. It will continue because cloud computing has not yet reached what we call the maturity phase of the product. Another interesting data point is cloud computing, and that comes from Forbes, is projected to increase from 67 billion in 2017 to 162 billion in 2020, with a growth of about 90 percent year after year. What does this mean? Why do I present you these statistics today? I do that because this indicates that this is a hot course. This is a course at understanding the cloud computing will be very important later on once you get a job and if you work in this computer science industry, because many companies are actually using cloud computing or ROS, or otherwise, are moving towards cloud infrastructure.

>> Let's discuss today about the utility infrastructure model. The traditional infrastructure model is that we have some forecasted infrastructure demand. That means at how much infrastructure we're going to use as our company grows, and therefore, based on that, we'll make some capital expenses. Let's assume that we buy today about 10 computers, all of a sudden everything works fine, but in a few months from now, our company grows a lot now we need 20 computers. Now, we buy another 20 computers. Then our company grows even bigger. Now, we actually need 100 computers and now we need to purchase another 100 computers. What we see is that we have situations where we actually buy computers or buy infrastructure, or storage, or whatever this is in advance from actually using that. What this means is that we have a significant surplus. We spend more money than what we need. This is a problem because that means that we're not really focusing or following the actual growth of the infrastructure itself. But the demand also is important. We may need 10 computers, we may need 20 computers, and 100 computers later on as the growth increases, but we don't use a computer all the time. We don't really store all the files all the time. We actually may move them in another storage medium. In many occasions, some of the traffic patterns follow what we call the [inaudible] traffic patterns. The actual infrastructure demand is totally different. What this means is that we even have a surplus. This is actually unacceptable because it costs the company extra money to maintain that. We also have a deficit. Occasionally, we may need more and we actually need a little earlier on, and because we didn't have the time to purchase this infrastructure, we are actually waiting to get this infrastructure if we don't do it properly. Which means that now we have a problem of a deficit of resources and therefore, scheduling this becomes a very complicated problem. Now, let's think about the utility infrastructure model. For example, this is what Amazon thought initially when they created cloud computing. They have this pattern where during the night we don't really use this, but during the day we actually use some infrastructure. Then what you see here is, we see this actual infrastructure demand. If we're able to book resources as we need them, then all of a sudden the surplus and deficit goes towards zero. Because you're actually following this infrastructure demand. In the scale up and scale out of resources is what cloud computing is comprised of. The ability to be able to scale resources as we need them. [NOISE]

>> Previously we have discussed how we have moved from being able to track the growth of our infrastructure needs, of our platform needs, and capture that based on how Cloud really does that with scale up and scale down of resources. What do we mean by scale up? Scale up of resources means that we can increase the number of virtual machines or servers that we deploy according to the needs of the applications. Let me give you an example. During the night, we may have more traffic at Netflix because people watch more movies. Therefore, we need to scale up more servers that common at these growth. During the night we have less users, but we can also accommodate the spare capacity to do some video and coding of our videos, for example. Same thing happens with many other companies like Amazon. Amazon, when people go and buy products, let's say on Black Friday, they scale up all these amazing resources to be able to come on this traffic, but the next day, on Saturday or Sunday, or even next Monday after Thanksgiving, there is not much traffic or as much as Black Friday because of the deals they offer. What happens is now they have to scale down these resources because they don't really need them. The model they created is, well, something, if I don't really use them, I can offer them to somebody else to do that. That's how Amazon Cloud came up. Now, let's see the history and the momentum from mainframes to the [inaudible] of things. In the beginning, it was the mainframe and the terminals. You used to sit down and work on a big, beefy computer. You can't do really much, but you could really type and do a few commands left and right. It devote users do individual work by connecting to a central computer, you did your work, and somebody else came and did his work, and somebody else came and did his work. We used to have all these disks and cassettes, we then plug in and play and to store the data and so forth. Then [inaudible] about 1980s, were talking about the personal computer. Now, users did individual work on their own desktops. You sit down, now, you have your own, at your own system, at your own house, and you'd really plugin and diskette that, you store the data and move to another computer you put the diskette in, and then now, all of a sudden you have the data, you just moved from one location to the other. The problem back then, it was of course, that if you had a magnet, data could be lost. There were many issues back then once you start moving data left and right. Then what happened about 1980s, we're talking about the network. Then the PCs got connected, and now, you create the network of PCs that they talk to each other, you have a centralized storage system, you have a centralized printer when you print everything and so forth. Now you're creating some security because you don't want invaders from the outside to access your network and therefore you get firewalls for outside folks. Then about towards 1990s towards 2000, then there was the web revolution. Now we had all these web application and Google started becoming big. Facebook started being in all these comments that we actually know, Amazon, and so forth, became the big entries in the web. They created this web where people could actually go and retrieve information, connect to other computers and to other people, and really be able to obtain information without really needing to have a physical machine other than on laptop. About 2000, 2010 became really big. We had server had to become cluster of partial computers. Care and feeding become expensive and that's going to become very complicated in all over the world. Therefore, we had to accommodate this grow, this huge growth of the Internet. Nowadays, we're getting to the momentum of the Internet of things. Now were giving Internet to every single device around the world, but where this is like these vehicles, and assets, and persons, and pet monitoring and controlling, this is one. Now, we have agricultural automation. We put sensors sourcing in the fields and we let our tracks go around the fields and effectively be able to monitor the field remotely. Now, you know how much water you can spend in each of portions of this field because now you have data points from all of the sensors you have deployed. Now, also have energy consumption. [inaudible] came up with this intelligent bulbs where they contain scholars and they can do that from a mobile phone. You can change, for example, other things like how many lights are on. You can change them based on feedback from your TV. Having connected items in things that they interact based in changes. We also have security and surveillance. We have cameras all over the place, all of them connected to the network that they're able to monitor. Building management were now we can manage a building from a central entity like elevators, we can control them from a smart phone, or water, we can monitor that. To telemedicine in health care where we can do surgeries remotely or we can have experts from different areas really do big parts of monitor. Let's say patients remotely without really needing to be there. To smart home and cities where, for example, now, if there is high traffic in one intersection, we can really route people [inaudible] by changing the way traffic lights operate. To everyday things from shoes. Now, Nike offers the essentials that you put in your shoes so they can know how many steps you do and how many calories you have burned. Then mobile to mobile wireless sensor networks and embed mobile things. A lot of things is now becoming the fact of things that brings the Cloud computing a huge value.

>> [NOISE] Now let's move to the datacenters and supercomputers. This is the ECE 530 cloud computing. Once the website becomes really big, and we've seen that from the Internet of things and really, really important. Now what we have is several pieces how to live in expensive datacenters. This is an example of the Microsoft Data Center in Dublin. It's about 27,000 square meters, about 22 megabytes and cost about half a billion dollars to build. As you can see, it's very expensive to Gaza for a lot of cooling and a lot of real estate to build a datacenter. Microsoft really needed that because now they have to serve customers for other applications where there's emails, or Hotmail, or accessing the web, and so forth. Now we came to mega datacenters like huge datacenters, huge facilities. A lot of people to be employed to be able to serve the need for that. [NOISE] After that, we required a lot of things like now cooling an infrastructure to build and a lot of batteries and backups to be able to maintain the infrastructure whenever it really something was going south. All this infrastructure became really important. The challenges to that is how do you become agile, green, and smart growth for that. You can really move datacenters around. It's really hard and you need a lot of expensive materials and employees to be able to build and maintain infrastructure like that and really scale up for the growth. In summary, datacenters will require a lot of electricity power like 1.5 percent of all US electricity back in 2007, later alone 10 years has RAM at where we are right now. It was a long lead time to build. It require about 3 years, or 4 years, or 5 years to really build a data in up and running. All this time you don't have an infrastructure available for you, which means that you are watching without it. It's an inflexible investment of capital. Once you put your money in your data so you can't get them out. It's not like a bank where you can invest it. It's more about you're putting your money into infrastructure once you really need it. It also requires very specialized skills for infrastructure like security, failover, and load balancing. These skills are expensive to buy them. Once you have these people that have these skills, really keeping them around, you can only keep them for a specific amount of time. At the same time, you get a small common, let's say you're now building the new snap or the new application is going to change the world like voice of our API or whatsoever. It takes out time from your core competencies. If you start being datacenters, you're really doing what's important for you. Of course, it's hard for all but largest communist to really own and run. If you're Facebook, Google, Amazon, Microsoft, you can really build datacenters because you have the capital like an Apple. But if you're in a small midsize company, really spend your money in datacenters is not really something that you really want to do because that means that you're not spending them in core competencies. Let's now think about how these momentum started. What is a supercomputer? A supercomputer is a collective of computing system that provides processing probabilities exceeding that of any system ever built. The process of built means are built quickly perform calculations. You can collect and process tremendous a lot of data and unique habits of computationally challenging problems. Supercomputers, of course, will be the first level or the first way, the father of God for behold, call it. Here's an example of the NASA Supercomputer, which is really huge like a big datacenter like now NASA really needed that to analyze a lot of system. That thing was built in 2003 and has 10,000 processors, one terabyte per second memory bandwidth, huge teraflops of performance. IBM Roadrunner was another big supercomputer in IBM for every significant amount of time, invested a lot in supercomputers, not so much anymore. It was the second fastest supercomputer that was built in 2008, 1.38 Petaflops of data, 24,000 processors. Of course, you really need like advanced scheduling system to use this supercomputers because otherwise we're having lower your resources, your electricity, cooling systems and all this infrastructure build really spent for nothing if you don't use them at all times. I would say the most modern one and the most beautiful one is the one Barcelona in Spain they called MareNostrum. MareNostrum is Mediterranean seen Lottie and effective are the most nicest ones. It's not like huge. It's about 48,896 Cores. It has about 925.1 teraflops in the linpack performance. Theoretically, it can achieve more than 1,000 teraflops. It burns about one megabyte per hour. It has InfiniBand FDR, and it's based on Linux. If you look at the networking. The networking of any datacenter or supercomputer has to be very clearly defined. You can have cables left and right because it's hard to maintain them. It's like you can see you have thousands of kilos coming out from different places. As you can see, this is a great organization of one of the MareNostrum Interconnection Network. The question always comes, what's the fastest computer out there? The fastest really entirely it's called Tianhe-2. Tianhe-2 or MilkyWay-2 is a clash of Intel Xeon E5-2692 with 2.2 gigahertz CPU but Xeon processor. It has about 3,120,000 cores, which brings about 33,867 teraflops. It's about theoretically speed of 54,902 teraflops per second. We're taking about 1,024,000 gigabytes of memory. This is a huge investment in really doing advanced processing for biological analyzing DNA or doing any processing that is important to be able to scale up for the computational needs that require none, right now based on the big data growth. As I said, super promoter is the grandfather of Cloud computing because we have so much resources out there that we don't use them all the time.

>> Welcome to the cloud computing and today we're going to discuss about cyberinfrastructure. One of the fundamental elements of how cyberinfrastructure and cloud computing and many other pieces started was effectively the high-performance computing. What is high-performance computing? The application of computer technology exceeding the capabilities of a desktop computer to solve problems that cannot be easily solved on a desktop computer. If you start thinking about the definition of high-performance computing, the way we just defined it, it has a lot of similarities in the way cloud computing is being used. What I mean by that? Now, we don't have enough resources in our own systems, therefore, we have to use a certain pool of resources. To be honest, high-performance computing, HPC has the scheduling of resources. It's pretty much the same as in cloud computing. Let's see now what does a really cyberinfrastructure includes. It includes; supercomputers, cluster computers, distributed computing, grid computing, and any kind of exotic hardware. For example, recently for machine learning and artificial intelligence, we've seen specialized hardware we've based on FPGAs, that is, field-programming gate arrays. [NOISE] What is grid computing? Grid computing is a high-performance computing system built as a geographically-distributed system from a collection of cluster computers, supercomputers, desktop computers, and storage systems, all collection via high-speed network. This is a definition of those two first. Now, what is overall, the term cyberinfrastructure? It's an information technology framework for discovery and collaboration. More specifically, it is anything related to instrument and sensors, high-performance computing, cloud computing, data storage systems, visualization facilities, all tied together by a high-speed network. In other words, cyberinfrastructure is pretty much anything that is outside of our own laptops. This is the infrastructure existing to help us analyzing data, store data, and so forth. [NOISE] To be honest, the data for cyberinfrastructure came out from the federal government. About 2000-2001, NSF, The National Scholarship Foundation created this terminology. If we start thinking about cyberinfrastructure, that is anything related to all the components that we use to do things like high performance services, data, information, knowledge management services, observation measurement, fabrication services, interfaces, virtualization services, collaboration services, all tied together and connected by a network, operating system and middleware. This is effectively the cyberinfrastructure and includes both the personnel as well as the hardware, the software, the services, and the organizations. The reason we tried to define cyberinfrastructure, it's because it's a term that is being used out there and sometimes, it's getting confused with cloud computing. Cloud computing is part of the cyberinfrastructure.

>> Let us now see the benefits of Cloud Computing. The solution to the problem we have mentioned in our earlier lectures about the data center is to outsource the data centers. The reason you want to go do that is because effectively data centers does not help with columns of scale. You have this long investment that you do in a specific infrastructure that you cannot really move it around when you want. But with Cloud Computing, we mentioned before about how we can really follow economies of scale, like as we grow, our capabilities of computers also grows. Because of scale, now we can afford specialized skills. We don't have to spend [NOISE] our specialized skill salaries into things that are hard to move. Moreover, we see more and more applications getting really big in the website like Facebook, Google, Netflix, Amazon, and so forth. Those web developers can now concentrate on their core competencies instead of giving them and give them a market advantage. Instead of focusing on, let's say, deployed data centers. At the same time, the right time to start from having an idea to having a productional system, is really, really small nowadays. In addition, you don't really need a lot of infrastructure to support the computing resources in the beginning, because now you have lower capital requirements. At the same time, computing power becomes a commodity as the electrical power in 20th century, and it's very, very important. The benefits of Cloud Computing, it can reduce capital and operational expenses. You no longer require to make a large upfront capital investments to data centers. You eliminate need to plan ahead for provisioning and you allow communists to start small increase as the move ahead and as they go, which is the pay-as-you-go model. [NOISE] You simplify application development management by common programming models across mobile, browser, client, server and cloud. Access to strong ecosystem of widely deployed applications and now you have integrated with existing ideas like software or assets. The Cloud benefits; The Cloud offers low pay-as-you-go pricing with no upfront expenses or long-term commitments. This is important because now you don't have to have any other commitments, or you don't have to have any other SLAs or SLOs. Cloud vendors build and manage global infrastructure at scale. They pass the cost saving benefits onto the clients in the form of lower prices. What we have been seeing in the last few years running private Cloud providers, is that Cloud providers let's say they data CPUs, data networking and data storage, and because they do that at massive scale, the cost on a per entity or per a company, is way, way smaller than it would be to update your own infrastructure to the latest, let's say CPU and so forth. With the features of Cloud scale and expertise, cloud vendors have now been able to lower their prices. [NOISE] Now, once you guys go out in the market, and to become Cloud engineers, you have to make into consideration as well that this is not your data center, and maintenance can happen without you knowing anything. Your instances will die. You have to use certain resources, and this is a problem. Now, your resources are not just used by you, but by many other entities. You're setting those resources. The architecture may change as well. You don't really own it. But even if you owned it, architecture evolves over time and over the years, and therefore changes are inevitable and of course, you never see the light. The common idea to have when you use the Cloud is that you must not trust it. You must assume that any failure can happen in the Cloud, and failures will happen because it's not all your own resource, but failures happen also in data centers. It's the way Cloud engineer have to think. They have to think that failures will happen, instances will die, availability zones or less a dataset may go down at any point of time, and therefore, they have to create systems that will be able to scale and as well as handle situations with failures.

>> Welcome back, let's discuss today about the common Cloud vendors. Amazon Web Services is the most popular web Cloud vendor out there. AWS Web Services was actually one of the first, they offer different types of services for compute, to store adds to a database, to networking, migration tools, developer tools, management tools, security and compliance, analytics, Internet of things, mobile services, application services, and even machine learning, they do offer. Their services are fairly mature, which means that they are other level where it's being used in production by numerous companies, and there are also numerous companies that run the whole ecosystem on Amazon Web Services. Definitely the most mature Cloud vendor out there and one that has mostly the largest portion of the Cloud. If we see the AWS named as a leader, AWS has been named a leader in the infrastructure service magic quadrant. You can see now the leader in visionaries, Amazon Web Services, he is the one that is better to execute, as you can see on the left side, as well as it is more complete in terms of their vision, it is better situated than what Microsoft is. Google also comes third with being one of the leaders also, the three leaders in terms of the infrastructural service. Three smaller infrastructure services are even smaller than that, like Alibaba, Oracle and IBM are still trying to find a competitive advantage in their market share, but to be honest, all these other three have started way earlier on. It's really hard to get into that market. AWS officially launched in about 2006, we're talking about 12 years of experience. It started off by offering IT infrastructure services to businesses in the form of what we called Web Services. Amazon Web Services has data centers in US, Europe, Brazil, Singapore, Japan, Australia, and many other places as well. AWS provides highly available, scalable, low-cost infrastructure platform in the Cloud that powers hundreds of businesses in more than a 190 countries in the world. The beauty about AWS is that once you alliance with AWS, you can be able to be accessible in 190 countries in the world, which means that you can easily get from zero to a global business fairly fast. If we see the history of AWS, it started by Benjamin Black and Chris Pinkman, where they presented these paper that they envisioned Amazon retail computing, being automated, standardized, relying on web services. What we knew back then actually as Amazon selling books, that's when Black and Pinkman thought about having web services that they can support that retail business. In about 2004, the first service called Simple Queue Service started, it was launched, and then in 2007, Amazon had about 180,000 developers on the platform. 2010, all of Amazon.com Services moved to the Cloud effectively. They became one of the first Cloud native companies. 2011, there was a major outage on AWS because one of the elastic block storage volumes was stuck, and unable to take reads and writes. Since then, we're talking about seven years of progress. EBS has been fairly highly available, no single point of failures, still exist right now. In 2012, there was the first re-Invent conference. Re-Invent has evolved as a major conference about Cloud computing with more than 40-50,000 people attending the conference nowadays. It has been held every year in Las Vegas, in about end of November timeframe. 2020 AWS and Amazon promises that all the Cloud would actually run on renewable energy. Another big Cloud vendor, as we discussed before, is Google Cloud. Google Cloud is growing really fast. They have the App Engine, Compute Engine, Container Engine under the compute, they have Data Storage, Bigtable Cloud Storage, Cloud SQL and Cloud Datastore, on big data, they do have a number of products enormously later gum arabic products. If we try to compare and contrast these two services, I would say that Google Cloud services develop at a fairly more complete when they come out in the market compared to what you see on Amazon, but on the other hand, Amazon has the most services out there compared to any other vendor. Just a very short history about Google Cloud, 2008, about two to four years later than Amazon, the Google App Engine announced it's preview. It didn't come to production until about two years later to three years later, she was talking about a gap of a few years between Amazon and Google, Google Cloud Storage as large as 2010 and 2012, the computer engine is lapsed, AWS launched 2006 affecting the computer engines, talking about five to six years of experience that Amazon has built in a highly growth area. Definitely Google has a lot to cover, but I think they're growing really fast and really serious in this game. Another big player, maybe the second biggest player is Microsoft Azure. They do also have a number of services like compute, web, mobile developer services, integration, analytics and data, and then all the services they offer to the client in the same way as any other Cloud vendor does. Definitely mature service as well. They started about 2008, they announced the Windows Azure platform in 2010, it became commercially available, we see similarities with Google and Microsoft in terms of the timeline. 2014, there was a major outage that effectively broke MSN.com, since then things have improved, highly available services Cloud vendor has to be highly available because single-point of edges cannot exist in the Cloud computing domain. Overall great Cloud vendor. Now, the way we think the future, we don't know what will happen in the future, but what we do know is that right now, Amazon definitely has a competitive advantage over any other Cloud vendor. In the future, we'll have to see about how these things evolved over time, but then we see some good execution both from Google and Microsoft, and we'll see how that goes. What I believe is the future of Cloud computing is, the infrastructure as a service has been already commoditized, that means that having a virtual machine or a container to run your systems is easier to do it on the Cloud than on your own laptop nowadays and cheaper probably. We will see now

the platform becoming commoditized for example, I want to store my data. I don't need to run my own database. I can really host everything on the Cloud.

>> Let us now discuss about what happens if the Cloud dies. Like, what if people are using Cloud and all of a sudden, the vendor that is offering the services decides not to offer services anymore, which means that, that may affect other commodities as well. The first thing that you have to consider is that the Cloud probably has better availability than you could do on your own. However, we have to consider retaining as much as in-house capacity as you need to stay alive and muddle through. That means that you need to have each extra capacity deserved from the Cloud to be able to accommodate cases where you don't have high availability. Let's see an example, a hospital or police department, which gets electricity from grid for normal operations, but they do keep backup generator for vital functions in case of an outage. That means that you still may have some local deployments that you can run in the Cloud. However, to be honest, most big vendors that are Cloud-native, don't even have this extra capacity. Also, we must be careful to maintain ability to access Cloud such as local routers. Which means that it's not only about the Cloud being available, but the access to the Cloud by the local network has to be made as well, which means that our local routers and infrastructure to connect to the internet and the Cloud has to be there. Let's discuss now about ultra-sensitive data. Some core, vital data you just can't trust to anyone else, for example, a Fidelity account contents or US Department of Defense submarine locations. You can't use an external cloud, but you may consider deploying an internal cloud with specific safeguards. This is, for example, how most of the most important US Department of Defense accounts work right now. Of course, you don't want to upload your vital data to the cloud, especially to public cloud infrastructure, but you may consider having some of this data in a private cloud infrastructure. These companies, like the Cloud vendors, often have much larger stores of data with low-security requirements for which cloud could be highly appropriate. Let's see an example, Fidelity fund prospects, and reports, US Department of Defense purchases of coffee and underwear. This, you can hide in a public cloud vendor bearing in mind that those cloud vendors offer very high security, probably, high security than what you see in most commercial environments. Let's now see some legal aspects. Sometimes law requires that certain data to be stored in specific countries or location, for example, EU has the specific laws. Sometimes you want data stored in a specific location to be accessible only by specific jurisdictions, for example, you don't want your self-care data to be accessed, the United States healthcare data to be accessed by a company, let's say in Panama. Technology changing faster than law can keep up. This is a big problem because the law usually is reactive and not proactive. More than a little bit tricky, Cloud could hurt, hosting not available in required jurisdiction, or help, quick switch of hosting into newly required jurisdiction. The stance of the law happens usually easily as we said reactively. Now let's go and discuss the availability of cloud resources. How sure are you that your cloud provider will have enough cloud resources available when you want to scale up, let's say particularly in burst situations? For example, if there is a hurricane and you have a lot of users accessing weather.com. How do you accommodate the fact that the cloud vendor will have these sources? The other question is, how badly will it hurt your business if you want to scale up, but you cannot? Scaling up is not always an issue of the cloud provider, but also on how fast you can do that, and how many resources you have in the background to be able to do that. Another question is, what remedies are available from a cloud provider if you cannot scale at the time you want to the degree that you want? These are usually described in an agreement that you have made with the cloud provider, which is called a service level agreement. Amazon has an interesting idea of spot market for computational resources. That means that you do have a reserved resources, there is a spot market, and there is also a general market, and into each of these markets comes with a separate price for using the services. This is the end of the Section 133.

>> Let us discuss today, what is the cloud. We've been discussing about the cloud computing and how we want people to start thinking about migrating from data centers to the cloud. What is really the definition of the cloud? The basic characteristics of the cloud include the service models. These are divided three categories, the Infrastructure as a Service, the Platform as a Service, and the Software as a Service. Then we do have the deployment models, which can vary between public, private, community, and hybrid. These are the common characteristics. The cloud computing common characteristic is that the cloud is a massive scale. That means that you have many companies, and we need huge infrastructure to support these enterprises. You have homogeneity, you have virtualization, that means that you are supported by virtualized hardware, so you don't really touch the hardware itself when you use the cloud, but that you are touching a virtualized environment. It is based on low cost software. That means that you build on top of low-cost software. It is resilient. That means that if something happens to the cloud, the cloud doesn't really go down. You try to avoid what we call single-point of failures, though it is a more resilient infrastructure. It's geographically distributed, and the reason it is, because it has to be resilient. For example, [NOISE] cloud can be deployed in multiple datacenters. One can be, let's say US East or US West, or EU, or Asia Pacific and so forth. You try to distribute your cloud into multiple datacenters such that if there is a localized issue, you can go around it. It's based on service orientation. We can discuss about, how do I now offer Infrastructure as a Service? How do I offer Platform as a Service? How do I offer Software as a Service? Of course, it provides advanced security. The reason it provides advanced security is because if you want to really trust the cloud, you have to be able to also trust the fact that somebody else is now holding your data and your critical infrastructure. Some of the essential characteristics is the On Demand Self-Service. That means that you use it as much as you want, and you can decommission or de-provision whatever resources you're not using. You need to have broad network access to access the cloud itself. You have a common resource pooling. That means that it's a common infrastructure with many resources and these resources are shared across many people. It is based on rapid elasticity. That means that when you really want extra infrastructure, you can really get that, and of course, measured services. That means that whatever you are actually using, you can have a metric built out of it, so you can really capture the progress, how much you use, how much you have to pay, and so forth. A few years back, NIST created this working definition of cloud computing, which is the essential characteristics we just discussed. The service model orientation, which is the Software as a Service, the Platform as a Service, and the Infrastructure as a Service, and then the independent deployment models that one can deploy the cloud, which is public, private, hybrid, and community. When we discussed about having an On-Premise datacenter, which is the first column here, you pretty much have to manage everything, from networking, to storage, to server hardware, virtualization, servers, database, security and integration, runtimes, and applications. You really have to do a lot of work. When we think about Infrastructure as a Service, what you really do is, you let the cloud provider provide you the servers, which run on virtualization, so virtualization environment, which runs on an actual hardware, which requires storage to store the data, and networking to connect those systems. The only things you need to manage, in this case, as a user of the cloud, is the application, the runtime, the security integration, the databases. Now when we move to the Platform as a Service, now effectively the runtime, the security, and database, are offered as is from the cloud vendor, so you really only now manage your application and how it runs and interacts on top of these runtimes and securities and database frameworks. When we think about Software as a Service we're effectively having a console a UI where we access a specific software. For example, if you have doctors, they go and access a platform where they can enter the data for the patients and can be stored in a centralized cloud facility. Effectively they don't have to create APIs, they don't have to do runtime, they don't really need to know what's the database in the background. The only thing they need to know is going towards a platform or a UI in which they can enter this data. Cloud computing, or in simpler shorthand, the cloud, as we refer it, also focuses on maximizing the effectiveness of the shared resources. Cloud resources are usually not only shared by multiple users, but are also dynamically reallocated per demand. You really need some resources, you get it when you need it, and when you don't need it, you decommission it so that either somebody else can use it or you don't have to pay for it. For example, a cloud computer facility that serves European users during European business hours with a specific application, let's assume this is email, may reallocate the same resources to serve North American users during North America's business hours with different application, let's say a web server. What we're trying to say here is that you can provision instances to serve your emails during working hours in Europe, but during those off-working hours you can serve North American traffic, which is a web service. The application doesn't really matter, the resources underlying like virtualization resources is what moves left and right.

>> Let's discuss now the deployment models for the EC530 Cloud Computing. The deployment models, as we said, can be some sort of a hybrid private, public, or anything else [NOISE] What is really a public? A public is a Cloud infrastructure used by many other entities other than your company and can't be hosted on off-premise like for example, this is AWS or Google Cloud or Microsoft Azure. But the same time you can have your own private internal Cloud, your own deployment, your own API something that you do not expose the outside network on-premises internally. When we talk about a hybrid Cloud, this is effectively some combination between public and private because you may want to have some functions in a public environment and some functions which are very critical for your operation, a private environment. We gave the example in the past about Department of Defense and probably countries computing the resources from a public Cloud vendor and AWS does provide specific Federal Cloud data centers. But the same time some of the data you have, you want to keep them in your private Cloud, in your private infrastructure. So when we think about Cloud computing and the natural evolution of the web, we start from compute and storage. I want to have these machines to do this workload and I want to store this data and then we'll move to the platform as a service, for example, now I don't want to just store the data, but I want to store them in a structured way. Therefore now, I have databases or I need some extra security infrastructure so therefore, I need to have some roles and accessibility to specific resources. This is again is a platform as a service. Then we move towards the applications and especially mobile is important, like Android and iPhone applications that you develop and these interact with a Cloud web ecosystem. If actually we were in that website which is just a merely UI, and these you hit a web service and other web service search with whatever you want to see. An example of that is Netflix. You have this website where you go, you log in, and then you see all the movies, and while all of this happens in the background, there are thousands of microservices running on the Cloud that will give you the recommendation, would run big data analytics for you, would do live and offline machine learning and artificial intelligence staff and so forth. This goes back to the discussion we had about software service being applications on-demand, platform as a service, being developer platform for creating applications, storage and compute capabilities offered as a service. This is the infrastructure as a service. Now, there are four deployment models. The first is the enterprise to Cloud, where you have the public cloud. As we say, you have an enterprise in a public Cloud, you have your internet that interacts between these two and the data that the users can access these resources. Then this is the private Cloud where effectively you don't have any public component, but rather everything is inside your enterprise. Whether this is in my own datacenter or I'm forcing up a private Cloud in my own datacenter. Instead of going through the internet to get access to these resources, now you can access these resources by using the Intranet, which is effectively the local network. Then you have community cloud, where you have some variation of everything. Where you have many people accessing some public resources in a public cloud environment. Then there are some other sources accessed, either public or private. There's also an enterprise component to that, and the reason we call it a community is because you have intranet and you have internet as well. So you have a VPN. You have a VPN endpoint where you can securely have an isolated network compared to the Internet and access those resources. Then now you have hybrid models. What do you have a variation between public and private, and enterprise and local networks and so forth. So this more complicated setting and requires a lot of thinking to do that and open APIs. Because really, how do you deploy application oblivious of the underlying infrastructure? Now, the beauty about that is that most recently they have been orchestration systems like Kubernetes is one of them, that allows you to really deploy across multiple Cloud deployment models. You can redeploy on-premise or Cloud and hi-tech infrastructure to the overlaying obligations. Many enterprises want to move that especially security enterprises where they still want to keep some security infrastructure, but still use the public infrastructure. So we see a lot of movement from public Cloud or private enterprises total with a hybrid model, so we'll see how that evolves as well. The rationale, if you have to be the rationale, when do I choose between public and when do I choose between private club? The first is the security and privacy of business data is a big concept. You cant say that I'm going to upload my data to some sort of a public defender and don't expect my data to be replicated across multiple regions. Therefore, potentially going to Panama or to some other, either Bermuda islands, where the legal framework over there is totally different and therefore you don't know how the data will be used. The other big problem is if I really use the public Cloud, I may be vendor locked in. If I start converging towards specific APIs provided by these cloud vendor and I have to do that, if I want to use value-added services, I may have the problem that cannot really change the Cloud environment. Vessel is required for your time performance and reliability are important. For example, in a private Cloud, you can just control the resources and therefore you can say," You know something, I'm going to spend more money in more network bandwidth to accommodate that." But when you have to open with a Cloud vendor, then the cloud vendor has to do that in the background and have an SLA, service level agreement with you about it. Cost savings of the shared module achieved because of the multiple projects involving semantic technology the company actively developing. Again, if you move

towards a more like a ladder model, a certain model, which is the public Cloud, you can really have cost savings from really deploying your own infrastructure. Now you can really share resources and do allocation accordingly. So you don't really need to use resources when you use a public Cloud. Cloud computing for enterprise. What should enterprise think about Cloud computing? Revise cost model to utility-based computing. You have to move out of the mindset that I have my own systems and I ran them now towards a more, I use CPU per hour, I use gigabyte per day and this is now my cost model. Your Chief Financial Officer, has to really now create new ways to capture cost. You also have to think about included hidden costs for management and training. Now if you want to migrate, you need really to train your employees and these employees have to adjust to the new setting. Different cloud images for different applications you have devalue them. You have to do performance testing for actual testing and figure out if everything is working the way you want. You have to prototype and learn because learning curve is important in the Cloud, it's not the same as running your own datacenter. You have to link it to current strategic plans of, let's say, sensory detection of disaster recovery. If you already have a disaster recovery scenario in the background, then you have to accommodate that for the Cloud resources as well. Again, private-public hybrid services, infrastructure, platform and software. Now you have the users that interact with these services and videos can be edited from metrics to browsers, to laptops, to IoT or data, for example, you collect from sensors. They will go to the Cloud be stored over there and analyzed and so forth. Now if you stood about this SaaS Maturity Model, initially was level one is AdWord customer, one instance per customer. That was initially the case where initially you were a tenant and you were asking for a resource. You had a one web service that is one instance, effectively talking to you. As we move towards a more configurable environment, now we know what we're going to provide on per customer. Then we will talk about a multi-tenant efficient system where now, for example, a tenant can access a web service which can be in a single instance. Now the more complicated is now you have a load balancer, that balance between multiple instances and then its tenants can be in any instance. So if you think about that, now you can start thinking about how you can sell resources behind load balances, and load balance is just a service. It's a DOM service effectively that just forwards traffic to the least loaded instance. This is how we have moved towards using our own systems, our own computer, towards using a pre-made system, towards all of us actually one computer towards using in the background, some load balancer to access those resources. Software as a service is a model of software deployment. When an application is hosted as a service provides customers across the internet. SaaS alleviates the burden of struggle of maintenance and support because now you don't really need to update it every time that version of the system, you don't really need to update the software itself, but rather the Cloud vendor updates the software for you. The terms we're using is platform as a service and infrastructure as a service, which is if actually underneath the software as a service.

>> So let us discuss today about virtualization and virtual machines. Virtualizations. You cannot physically touch the Cloud server machines. Hardware can and frequently will change. To avoid dependency on specific hardware, you generally write your server program to run on virtual machines. So virtualization is a momentum that has been started about 15 years ago. More and more people are seeing value in terms of virtualization hardware so they don't have to really work with actual hardware. They don't have to connect cables, they don't have to set up machines and so forth. There other things you can set up is virtualized virtual machines which is easily replaced because the only thing that a virtual machine really needs is a baseline operating system and a host operating system. So let's now see the stack, the Cloud Stack. So the infrastructure itself is the hardware, either the data centers that host the Cloud. On top of it, you have some virtualization virtual machine monitor or a hypervisor. Then, on top of that, you create virtual machines which have the operating system like Ubuntu or Windows and so forth. Then, you build Cloud services that run these virtual machines. Then, the Software-as-a-Service is you build the Cloud applications that level as those services that have been built on top of operating system that runs on top of a virtual machine. So virtual workspaces is an abstraction of an execution environment that can be made dynamically available to authorized clients by using well-defined protocols. The resource quota can be CPU, memory cell, or network cell and so forth. You can have software configurations like operating systems or provided services. So this is the visualized stack. You have the hardware. You have the hypervisor that manages the interactions between the virtual machine and the hardware and on top of it, a virtual machine has an operating system and an application. To implement on which the machine's virtualization, you need abstraction of physical host machine. So you don't really need to configure how do you connect the cables. Hypervisor intercepts and emulates instructions from virtual machines and allows management of virtual machines. So the hypervisor does the allocation between the hardware resources and the virtualized resources without the service or the application having to know about that. There are different variations of hypervisors like VMWare, Xen, etc. Virtualization provides infrastructure APIs like plug-ins to hardware to support structures. So this is a virtual machine hypervisor as you can see here. You're going to have different virtual machines and each of the virtual machine can run a guest operating system, which can be anything from Linux to BSD to Windows. On top of that, people run their own applications. So virtual machine technology allows multiple virtual machines to run on a single physical machine. As we said, the hypervisor can be Xen, VMWare, UML, Denali, etc., and the performance. Para-virtualization, like I said, is very close to raw physical performance. That means that the performance you get inside a virtual machine will match the performance you get with the physical machine. It's not going to be the same. There will still be overhead, there will still be some penalty. But more than hardware like Intel processors, they do have capabilities to slice and dice the resources on how you allocate them to virtual machines. Same with operating systems. They do have modern instructions that allow for virtualized environment to run really, really high performance. The reason they do that is because to be honest, most of the purchases right now, processors and drives and machines are run by big Cloud vendors because they want to deploy largest and more geographically distributed Cloud computing infrastructures.

>> So let us now discuss about the opportunities and the challenges that exist in the Cloud computing era. What is the purpose and benefits? Cloud computing enables companies and applications, which are system infrastructure-dependent to be infrastructure-less. By using Cloud infrastructure, you can pay as you need and you can pay as you demand. So all of us can save capital and operational investment. I mean all of us, both the user can save capital investments by really not investing in data centers, but as well as Cloud vendors can leverage this shared resources and allocate them accordingly. So client can put their date on the platform instead of their own desktop, PCs, or of their own servers. They can put their applications on the Cloud and use the servers within the Cloud to do processing in data manipulations. So Cloud-sourcing, why is it becoming a big deal? It is becoming a big deal because using high-scale/low-cost providers is beneficial. You can have anytime and anyplace access to the resources via some platform, a web browser, or some other platform. You can rapidly scale your resources with incremental cost and load sharing, and you really need to forget about hiring, and keeping, and maintaining local IT folks. But what are the concerns? These are the benefits, what are the concerns? The concerns is your performance and reliability are now governed by the vendor. In fact, that relationship of the performance reliability is governed by the service-level agreement that the client has signed with the Cloud vendor. You also lose control of the data. The data are not anymore in your own infrastructure, in your own systems, or web servers. But rather, they are hosted on the Cloud. So application features and choices are effectively chosen by the Cloud vendor based on what they feel are the need of their customer, but they are not always certain about that. You have to do different interactions between Cloud providers. Actually, there are not good interaction between Cloud providers and there are not standard APIs right now. So any service Cloud provider can provide their own APIs. For that reason, it's hard to move between one and another. Of course, there are private security and compliance and trust issues, but that actually is something bigger. So the use of the Cloud provides a number of opportunities. It enables services to be used without any understanding of the underlying infrastructure. Cloud computing works using what we call economies of scale. We already said that. It potentially lowers the outlay expense for startup companies as they would no longer need to buy their own software or service. The cost will be on-demand pricing. Vendors and service providers claim costs by establishing an ongoing revenue stream. Data services are stored so you can access them at any point, at any place. In parallel, there has been backlash for Cloud computing. The use of Cloud computing means dependency on others, and that could possibly limit flexibility and innovation. The others can be either companies like Google, IBM, and so forth. Some argue that this use of supercomputers is a return to the mainframe computer that have been seen was a reaction against. Though, in my opinion, this is a false argument. Security could prove a big issue. When you go through all the security, you know which pieces of your infrastructure are secure and which are not secure, and you could work on that. Now, it is unclear how safe it is to outsource the data when using services ownership of data is not always clear. Like, will your data remain in that state or will be replicated in countries with different legal frameworks? In order to discuss about legal frameworks, it is important so that every country or every region has its own laws, and the way they enforce the laws is totally different. There are also issues related to policies and access. So if your data stored abroad, whose policy do you adhere to? What happens if that remote server goes down? How will you get access files? Of course, there have been users being locked out of accounts and they couldn't even access the data. Let say you forget your password. Now, you're locked out.

>> We're moving to the next section of the advantages of cloud computing. The advantages of cloud computing is the lower computing cost. You do not now need to have a high-powered and high-priced computer to run your cloud computing's web-based applications. Since applications run in the cloud not on the desktop PC, your desktop PC does not need the processing power or hard disk space demanded by traditional desktop software. Therefore, it reduces the cost to maintain a fairly advanced desktop PC. When you are using web-based applications, your PC can be less expensive with a smaller hard disk, less memory, more efficient processor. In fact, your PC in the modern world does not even need a CD or a DVD drive as no software needs to be loaded or no documents needs to be saved. You also get improved performance, because now you have fewer and larger programs hogging your computer's memory. You will see the performance for your computer to improve and therefore you can have other applications that are more like CPU intensive compared to using those applications that you get, compared to a really running all applications in your PC. Computers in a cloud computing system boot and run faster because they have fewer programs and process load in the memory. Reduced software costs. Instead of purchasing expensive software applications, you can get most of what you need for free. For example, you can get Google Docs for free. Google Docs actually, it is a Software as a Service. You could get a lot of applications of the cloud as well for free, or for very low cost. Better than paying for similar commercial software which alone can be justification for switching to cloud applications, so that Google Docs with slides and documents, and Excels, you don't really need to have all these whole Microsoft suite in your laptop if you can run it on the cloud. In a similar way, backups of Docs has Microsoft online, in which it does have a lot of these desktop applications online as well. Instant software updates. Another advantage to cloud computing is that you are no longer faced with choosing between obsolete software and high upgrades costs. When the application is web-based, updates happen automatically, available the next time you log into the computer. When you access a web based application, you always get the latest version and if you don't, then you can refresh and really get the upgraded version. You don't need to pay or download an upgrade, you don't need really to have a high-performance, let's say network gig link to do that and so forth. So you definitely lower the cost and you are going towards more about thin client. So thin client means that you don't need to have big machines in your system in your home. For that, you need a thin client and all the sources are available in the cloud. This is exactly why you see now, like Pixelbooks offered by Google, like laptops, and exactly what you do, you have a Google documents, e-mails, and so forth, and even if you did, let's say to write some software, sometimes we do have web UIs where you can do that. So improved document format compatibility, that you don't have to worry about the documents you create on your machine whether it's compatible with another application operating system, because now everybody runs the same version of the cloud, they are adjusted to that. There are potentially no format incompatibilities when everybody is sharing documents and applications in the cloud, and you can also collaborate on the cloud as well with this, it is Software as a Service. You have unlimited storage capacity. Cloud computing offers limitless storage. Of course, you have to pay for that but definitely the cost is way lower. Your computer's current one terabyte hard drive is small compared to hundreds of petabytes available in the cloud. For example, many of us store our photos in iCloud if you use an iPhone, and many of us store our photos in Google Photos if you use on Android devices. Like Google Photos, the storage itself is totally for free. So this is a Software as a Service offered by Google, which you don't have to pay at all compared to storing in all your devices in hard drives and then having to have multiple backups and so forth. You also have increased data reliability, because unlike your desktop computing the cloud provider itself has all the backups and redundancy needed to maintain your data. So if a hard drive goes down or they have redundancy to make sure that you don't lose your data because you are accustomed to that. In a world where few individuals desktop PC users backup their data on a regular basis, cloud computing has become a more like a data safe computing platform. Universal document access. This is not a problem with cloud computing because you do not take your documents with you. Instead, they stay in the cloud and you can access them whenever you have a computer or an Internet connection. Documents are instantly available wherever you are, so you don't really need to copy or carry flash drive, or have to carry a DVD as we used to do in the old days, your data are located wherever you are as long as you have an Internet connection. Of course, you always get the latest version when you edit that you have document and you try to access it from another place. You don't really need to save and re-do it, your data are there whenever you open the platform. It's also easier to collaborate. Now, you don't really need to say, "I'm going to have this document that accompany your computer [inaudible] and give it back to me or email it to you." Now we can have this online platform where we can all go together, it has collaboration. It's totally collaborative document, you can have collaborate on slides, you can collaborate on Excel sheets, you can collaborate on coding, you can do a lot of different things, many users do this. It's definitely part of a Software as a Service platform. It is also is device-dependent. It doesn't matter if you use Linux, Mac, or Windows. If the system and the service is run on the cloud, then your cloud is just a gateway towards accessing the data. So you don't need to have an operating system, you don't need to upgrade it.

Your computer doesn't really have to happen in the cloud. So in the old days, we had to update software operating system, we had to do security patches, and so forth. All these are now taken care by the cloud itself.

>> So we discussed the advantages of cloud computing. Of course, they are not only advantages, there are disadvantages as well. It requires a constant Internet connection, it is impossible to access the cloud resources if you do not have a reliable connection. Since you use the Internet connect to both your application and documents, if you do not have an Internet connection, you cannot access anything even if you really own these documents. A dead Internet connection means no work and in areas where Internet connections are few or inherently unreliable, this can be a deal-breaker occasionally. So you have to get your system in some sort of a hybrid mode. [NOISE] Of course, it doesn't work well with low-speed connections. If you use dial-up services which, again, are probably extinguished right now, but still if you use your phone and you're tethering, and your connection is not really great, accessing those cloud computing resources to be able to interact with a cloud computing platform is fairly difficult. Web-based application requires a lot of bandwidth to download, as do really large documents. [NOISE] Features might be limited. So for example, in mostly desktop, laptop applications, you can have a lot of features and added things because it has been years of development. Now as this move to the cloud, occasionally you may be bound to specific features or maybe the corresponding online application may not have the full features as the desktop based one. For example, Microsoft PowerPoint has way more features that the corresponding Google Presentation web-based offering. It can be slow. Even with a fast connection, web-based applications can sometimes be slower than accessing a similar software program in your desktop PC. That is because you have to pay the network penalty to reach this application. Everything about the program, from the interface to the current document, has to be sent back and forth between the computers in the cloud. If the cloud servers happen to be backed up at that moment, or if the Internet is having a low slow day, you cannot be get instantaneous access from desktop applications. Stored data might not be secured. With cloud computing, all your data on the cloud. I mean, most probably they are secure, but you have to trust somebody else with that. Therefore, how can you trust somebody with confidential data is the question here. Stored data can be theoretically lost. For example, if you lose your password, now you don't have this data. Or if you don't have physical or local backups, sometimes things may get bad. Because let's say somebody access your account, deletes your data, what do you do after that? So data can be lost definitely. HPC systems. It's not clear that you can compute-intensive HPC applications. So you cannot really say, "Oh, I have a high workload, I'm going to write it right now on the cloud." You really need to have all these resources pre-scheduled. But if you have a piece of [inaudible] home with a VDI graphic processing unit with advanced GPUs, you can really do that in your system. Nowadays though, to be honest, the more expensive systems are moving to the cloud, like [NOISE] types of applications on Google Cloud are super fast, so you don't really need to have specialized hardware on your system. Of course, scheduling can be tricky because now you have to schedule your really high-performance workload, and you have to wait the other resources to be freed so that you are able to do that. In general concerns, each cloud system uses different protocols, different APIs, may not be possible to run applications between cloud based systems. Amazon has created its own database system, not SQL 92, and workflow system, or this is called Aurora. Now, you may not have backward and forward compatibility.

>> So let's go and see an example of an infrastructure as a service. So we'll start from the lower level of the Stack of the Cloud computing, move forward in the platform, and then the software as a service. But let's start with an example of infrastructure as a service. So the most well-known infrastructure service is probably Amazon Web Services. You go there, you reserve an instance. Same with Google, same with Microsoft. But apparently, a number of companies have agreed to create a corresponding open-source language called OpenStack. So OpenStack is an open-source Cloud computing platform. It is an infrastructure service solution. It supports many independent services. OpenStack was a joint Cloud computing project. It started with Rackspace and NASA and aimed to provide a new infrastructure as a service. It lands in 2010. Today, it's a community-based supported open-source project with about 15,000 people, 136 countries, and more than a multiple hundreds of companies supporting it. So OpenStack is based on Python, so it has more than one million lines of code of Python. More than 200 companies support it from Ubuntu, Rackspace, HP, Red Hat, Cisco, Intel, IBM, SUSE, VMware, Dell, all these companies that effectively will take them significant amount of time to deploy a new infrastructure service. They came together and collaborated to create OpenStack. So the high-level architecture is we have the central dashboard, the same way as you have in every kind of Cloud computing service. Then you have independent services where this networking service, your block storage service, your compute service, your image service, your object storage service, or your identity service. So dashboard is where you actually see all these resources and identity is the sources. Now, to be able to access these resources or even to view them in the dashboard, you have to have these privileges or the authentication. So identity effectively feeds each of the services so that we can define roles and access capabilities for specific users for each of these specific services, and all these services now go back and present what they have to do in the dashboard. So definitely, OpenStack is being a very, very interesting project compared to a lot of what we have seen. Now, the networking service as you can see feeds the compute service because now, if you have an instance, you need to configure a virtual network interfaces. Same with block storage. If you have an instance, you want to attach block storage so you can save your data. Then the compute service. Once you store the data in the compute service, now you can back them up in an object or in the object storage so that's why that computer feeds these other services. So anyway, this is the genetic architecture of OpenStack. Definitely an interesting project and definitely an evolving idea.

>> Let's go back to see now the OpenStack Services. The OpenStack Services are multiple independent components and multiple independent services. The first one is the compute, instead it's called nova, that's the code name. It's an on-demand network of virtualized machines, AP is like EC2 instances in AWS. KVM and Xen are the two available choices for hypervisor technology or you can even use containers like Docker. As you can see in the picture below, you have the instance name which is called test, then you have the operating system that you are adding. Then the IP address you have allocated the size, what type of virtual machine this is. The keypad if, for example, you have a specific key that you want to access [inaudible] mode. The status if it is active or inactive, the availability zone, and the task and whether it's running or down and so forth, and how many minutes has been up. Then you have different steps, different axis. I create snapshots and so forth. This is what you see in the UI when you provision an instance. Network, which is the network surface, the code name is called Neutron. It allows users to create their own networks and then attach interfaces to them, it is highly configurable, and it has a plug-in architecture. It gave about of the legacy networking called nova-network, but that actually was deprecated. So nova-network goes mostly like how do you collect instances, but then they decide to have a separate network project. It's simple and it had lack functionality like VPN, load balancing, and firewall. Neutron does have all these capabilities now. Cinder which is the block storage. Block storage is effectively the way that you attach a volumes to your instances so you can save data. It's like a network attached volume. It provides persistent block level storage devices that you can use an OpenStack compute instances, and these have to be attached effectively to any instance of virtual machine. The project itself, the UI, you can manage the creation. They attach and detach of the block devices to those instances and servers. You can see that here in the UI, for example, you have a volume of one gigabyte with dev test1, it's currently used and it's attached to test WAN on this specific mounted or this specific folder and on the specific availability zone. The next project is called swift, it's for object storage. This object storage is effective when it's not a network attach thing, but rather you have to call an API to access, which means that it's usually used when you want to backup data. So let's say I have this backup, I call an API or backup this data and then this goes in a specific container. It accepts file to upload, modifications to metadata or container creation. The swift architecture is very distributed in order to prevent single point of failures and you can scale it horizontally. For example, you need more storage space, you can just add it and, of course, if you backup the data, you don't want to lose them. That file it has to avoid this single point of failures. Identity is the keystone, that's the name of the service called keystone. It provides a single point of integration for OpenStack policy, catalog, token, and authentication. This is actually the hardest one to configure because once you mess it up, you may not be able to access your resources. Of course, it supports multiple forms of authentication, including standard user name and password credentials, a token-based system, and AWS-style, user, tenant, and role. The next one is they both store the code, they glance. It provides discovery, registration, and delivery services for disk and server images. It can also be used to store and catalog at an unlimited number of backups. So this is more like longer term backups where you would store them and stay there forever. So this is the image store called the glance. Last but not least, the dashboard where you access all these resources. The code name for that is called horizon. It provides administrators and users a GUI, a graphical user interface to access, provision, and automate those cloud-based resources that you have provisioned before. So very interesting. Now, if you start comparing the resources and the code name using a nova like EC2, neutron like a VPC, cinder is like elastic block storage, swift is like S3, glance is like an AMI, horizon is like console, and keystone is like IAM roles in AWS. So just to give you an idea about if you've ever had experience with AWS, now you can actually relate those code devs in OpenStack.

>> Hello. Welcome to the second section for the ECE 530 Cloud Computing for the introduction to Amazon Web Services. Amazon Web Services is a subsidiary of Amazon that provides on-demand Cloud computing platforms and APIs to individual companies and governments on a metered pay-as-you-go basis. We have already discussed about how the Cloud works with a pay-as-you-go model, and this is how AWS also works. AWS is effectively just one of these many Cloud providers. Initially, it was launched in the July of 2002, mainly looking at the Amazon platform where people go and buy different types of products. They had spare resources that they didn't know how to use them, so they thought that we can use those spare resources of compute to give them to other users so they can use them on a pay-as-you-go model. In 2018, the AWS revenue was captured about 28.2 billion. As of today, the AWS is one of the major profits for the Amazon parent company. I think it's probably one of the highest growth subsidiaries that Amazon has. AWS technology is built on top of server farms, they call it datacenters that are deployed around the world. As of 2002, AWS comprised of 212 different types of services that they can deliver from compute storage, machine learning, and different types of services. Fees are based on a combination of usage like pay-as-you-go model as well as standard expenses. As of 2012, AWS is currently the dominant Cloud provider, capturing about 34 percent of the market. Obviously, currently with other providers like Google or Microsoft, the specialists may have shifted a bit, but still AWS remains the dominant Cloud provider by far, maybe three times larger than anybody else. AWS is a collection of remote computing services that together make up a Cloud computing platform. The three major elements, or high users elements of AWS is the compute. The service is called Elastic Compute and effectively this is virtual private servers. Somebody speeds up one of the servers and they use it as they would use their laptop, as they would use any remote machine that they want access to. So you get a system with a Linux operating system or a Windows operating system that you could use as you use your laptop. It's a virtual machine. Other compute platforms is the Elastic MapReduce, EMR, which is mainly a Hadoop framework running on EC2 instances. They provide different types of storage services. One of them is the S3, which is a Simple Storage Service. Simple Storage Service now has integrated with Glacier and other offerings where Glacier is the low-cost and long-term storage. Now, they have deep archive and have different types of storage tiers, as they call them, where you can effectively store your data with lower prices in terms of actually storing the data but with high prices when you actually retrieve the data in the low tiers. The basic tier S3, usually the access. It's a little more expensive to store the data, but retrieval rate is lower in terms of expenses. Different type of further services that they have is the Elastic Block Storage, which is effectively a persistent block level storage. So you see the difference, S3 is an object storage at Glacier and EBS is a block storage. It's a different type of offering based on the requirements. EBS is those storage devices that you attach to your EC2 instances that you can use to store extra spaces, just the local hard drives. Other offerings include databases where you can store structured data. The major one is DynamoDB which is the most famous one, which is a NoSQL backed by Solid State Drives. They have different other offerings like ElasticCache, which is an in-memory based Memcached, which is a software that effectively people use to cache data so that they don't overload the databases. It also have different type of databases for relational workloads like the called RDS, Relational Database Service. They have different types of variations, MySQL, Postgres, and so forth. These are similar to what you would get like Oracle or SQL server and so forth. So the AWS platform effectively is comprised of the Deployment & Administration layer, the Application Services, then they use Compute, Storage, Database, and even other different types of services right now. Then below that, AWS has the Networking where it's connecting all these virtual machines and all these EC2 instances together. Then effectively under that, they have the AWS Global Infrastructure, which is a global deployed data service infrastructure throughout the infrastructure. That's basically the introduction to AWS. In the next part, we're going to discuss more detail about the infrastructure itself.

>> So let's talk about the AWS infrastructure. This is the Section 2 of the course. So AWS global infrastructure is comprised about multiple regions. They have effectively data centers around the world, from United States, Europe, Asia, South America. There's about 11 regions right now deployed. [NOISE] So those locations, they are allocated in different geographic regions, mainly the reasons we do that for disaster recovery. For example, if you have a hurricane that may hit one of the regions, then the other regions may be available to reroute the traffic. So effectively, each region has also, within that, isolated locations known as availability zones. For example, if we discuss about the US East, which is Northern Virginia, then you'll find out that there is effectively three data centers that are connect with each other, and they're separated by specific miles between each other. So even a region itself has three separate data centers. So that means that if the whole Northern Virginia goes down, you still have other regions. But if part of Northern Virginia goes down, we still have the other availability zones. So each region has two or more availability zones which are distinct data centers providing AWS services. I think the one region that has two availability zones is in US West 1, that is in Southwest. The major reason because it has two availability zones is because of very high real estate cost. So its Amazon EC2 region is designed to be completely isolated from the other Amazon EC2 regions. As we said this because of fault tolerance and stability. If one region goes down, if one goes down and you still are able to operate your service on the Cloud. As we see, as we deploy distributed systems on the Cloud, we'll identify that mainly this is the beauty of the Cloud, that somebody has taken care of the fail over, and the tolerance and the stability of it, and therefore, we only just need to develop an application on the Cloud that can scale by itself, at the same time be able to be available at all times with minimal cost. If you start to compare that, for example, if you have your own laptop or your own server at home, you can see that AWS or even any other Cloud provider provides substantially more stability and durability in cases of failures. So several services operate across availability zones. For example, includes like the storage service or database service, while others can be configured to replicate across zones. This is to spread demand and avoid downtime from failures. [NOISE] So let's go and see now how will you access website, how you can definitely go to different areas. So when you access a website, you make a web request and that goes to an Amazon Cloud front, that typically decides which location you are going to use. Right now, actually, this is probably outdated data, Amazon has at least 52 Edge locations. These are effectively Edge points and these are close to regions. So effectively, your request goes to one of these Edge points and then it's routed to the proper region to serve it. It used to cache data closed to the user, so latency are also reduced. So if you want to access through website a specific service, you can also cache a lot of the data within the Edge location as well. So effectively, they can be really, really fast. If we look at the AWS global infrastructure, we can see that there are different types of regions that each, as I say, that have two availability zones. One of them is Northern California. There are regions that are specifically designed for government, and then there are different types of regions around the world. Some of them that we see is actually coming soon had all the game. For example, India AP South 1 is already there right now. So if we see the AWS global map, we will see that it's divided in different areas. So when you say like US East 1, that covers United States East. There's US West Oregon, there's US West California, US West 2, and US West 1. There's South America, there's EU West 1 that is EMEA, and in Asia, Tokyo, Singapore, India, Sydney. This is how effectively AWS divides the areas that it sells through different traffics.

>> Let us now discuss the AWS networking. How is the networking services, and how's the networking being done in the background for the Amazon Web Services? So the number one service that is important to understand is the domain name system, so it's highly effectively unable to access website that is being hosted on AWS. The service is called Route 53, which is a highly available and scalable Cloud domain name system, DNS web service. Let's see an example. Assuming that the user wants to ask, "Okay, what is the website name that I want to see?" What you want to do is you want to resolve this name to a specific IP address. So what happens is the DNS resolver asks the AWS service called Route 53, which is the authority to give them the IP address that represents to that specific website. So Route 53 responds back to the DNS resolver, that the IP address for the specific abc.com is 192.0.3.2, and the DNS resolver gives this back to the user. So effectively this is how DNS works, you request for a specific website like www.facebook.com or amazon.com, and effectively you get to be routed to a DNS server. In this case AWS service have their own DNS server, which they call it Route 53. The second example is the virtual local area network, VLANs. AWS Direct Connect makes it easy to establish a dedicated network connection from your premises to AWS, they use what they call virtual local area networks. This is based on the IEEE protocol 802.1Q, which effectively says there's dedicated connection which can be partitioned into multiple visual interfaces. So therefore the data that you send over to AWS from your house, then it's actually encrypted and it is in your own network, so normally it calculates that bit and look at the data. Client can use the same connection to access public resources such as objects stored in AWS S3 using a public IP address space, and private resources such as EC2 instances running within the VPC using a private IP space, whilst still maintaining network separation between the public and the private environments. This is a great thing about AWS, is that you can actually split between what's publicly accessible versus what is privately accessible. Another thing that AWS provides is the AWS Direct Connect service, this is a service where effectively you have a direct connection to AWS, so you can have a very high throughput network connection. For example, here you may have a corporate data center, where you have your own system storage, movies or people at work, and then you use the Direct AWS Connect, which gives you about 10 gigabits per second and a 100 gigabits per second to AWS, which means that network to AWS is super, super really high, that's another service that AWS provides. Of course your network has to support that, but if it does support that then AWS make sure that your data are being rerouted to AWS really fast to access the services, so effectively get very high throughput and low latency by doing that. But the most important is to figure out is from your premises to the closest POP that AWS provides, point of presence means POP, with Direct Connect is to have high throughput. A virtual private Cloud is another offering by AWS, which lets you provision a logically isolated section of the AWS Cloud, where you can launch resources in a virtual network that you define. This allows you to totally separate your network with any other network, and therefore nobody can access, this is locked effectively. For example, you can create a public-interfacing subnet for your webservers, so you can have a public-facing app to Internet. But then your back-end services like database and application servers are only going to be faced by private-facing. So effectively, if you have somebody who's trying to access those, they could not be able to access, because they only get to access the public-facing. Let's give an example of a store that is hosted online, effectively you can access your own store to buy some clothes let's say, but the databases in the back-end systems that support the store are not accessible outside. It's important to maintain the security of your infrastructure as a customer of AWS, so AWS provides visibility to isolate your back-end systems and your front-end system.

>> So let's talk about AWS, EC2, and EBS service, Elastic Compute and Elastic Block Storage. So Amazon Elastic Compute is a simple web service interface that allows you to obtain and configure capacity with minimal friction. EC2 allows to provision virtual instances. Think about EC2 as the way that you would provision a virtual machine to do some work. So instead of using your own laptop, you're actually using now virtual machine that is based on a hypervisor learning on the system. The hypervisor that AWS is using is Xen Virtualization. EC2 now allows you to provision different types of operating systems from CentOS to Ubuntu to the Amazon's or Linux environment as well as the different varieties of Windows. Also for some companies where they want to provide their own operating systems or pre-configured instances, you can do that through the Amazon marketplace. So EC2 is effectively a compute service that effectively allows you to run your own virtual machines within AWS. So in EC2 you have effectively the route 53, which we discussed in the previous session, and then you have elastic load balancing. So when a request comes in that will be used by one of the, let's say web services deployed on EC2, then with AWS you can enable an auto scaling policy. That means that every auto scaling group that you have defined, can scale independently up and down and then you can have an ELB, an elastic load balance, that will distribute the load across those EC2 instances. So Auto Scaling. Auto scaling is effectively automatically sizing of compute clusters based on demand. You can define the minimum and the maximum number of instances in a specific group. Based on the data and based on the traffic, you can actually scale it up and scale it down. This can happen automatically by the Cloud service itself without the user doing any manual work. So as you can see here, we have an auto scaling group, and then you present it using this in CloudWatch. What CloudWatch is more like the monitoring metric system that AWS uses. Then based on specific metrics that you have defined, for example, increased or what is the throughput goes above 10 megabits per second, then effectively that causes up the scaling group to auto scale itself. So let's see now elastic load balancing. In this case, we have the DNS around everything. The user then goes to elastic load balancing, you have the auto scaling group that scales up and down the web services. So think about now you're accessing amazon.com on a day when everybody tries to access amazon.com, that for the service needs to scale up automatically. In the background what you do have is you probably have a database. You have, for example, the database of all the Amazon support the products that are related to amazon.com. Or if you're trying to access another vendor, it could be Facebook or the Facebook does really use AWS, then you will have the images that are of the faces of the people that are posting the images on the data store or even the metadata, and then you have the people auto scaling individual versus any other scaling group. Then of course, going back to the Facebook example, you will have the, for example, the customer data, the data store, but then you will have some of the images or some of the video files in the media files repository that will be set by the Cloud delivered network. So when we see these, think about these big [inaudible] this is being the media played. Same with Netflix as well. Netflix, you have the web services that access netflix.com. Once you get to stream a movie, the movie is actually being streamed by the media files repository to act specific on the delivery network. Amazon EBS is another service. It actually a block storage service. It's a service where you are mounting a specific block device to the instance so you can work on it. So for example, you can have an instance that has no hard drive, and AWS has a bunch of those. They have instances that don't even have a hard drive. So what you do is you can effectively atag or attach a block storage system that you can use to store your data, and it will appear as adorable back to point to the instance, no difference. So EBS provides a raw block device that can be attached with these two instances. EBS supports a number of advanced storage features including snapshotting and cloning. Now, the beautiful about the EBS is for example, if one of the volume goes down or if something happens to one of the volume, it actually does copy of the data, which is really great. Or you could even take snapshots of the data to another EBS volume or even to S3. So those are the things that allows you to effectively have high durability in the data in the amount points that you have on a specific EC2 instance. So let's see here an example. If you have Internet, you have access to your domain.com, it goes to elastic load balancer, you distribute the load across different types of instances. In this case, we have an emerge with I2.4xlarge. This is just an instance type out of the so many that AWS provides. Each of these instances now it's attached by a specific EBS storage. AWS may at some point provide a multi attached storage where we have multiple EBS instances that's the same instance, but for now, it provides the ability to have attach a specific EBS storage to a specific instance and all you add. Then you have an auto scaling group, which is effectively all the instances. As we said, as the load increases effectively what happens is, you vertically scale this AWS fixes. Now it's the 40, we have 10 to 80 or a 100. So another example is you go to netflix.com during the night hours when everybody is watching a movie, all over sudden, those web servers scale up to sustain to thousands to sustain the actual capacity that is being automatically calculated by the load balancer. In the morning when nobody is watching movies, then it scales down. Same with amazon.com and all the other services that follow the older pattern.

>> Now, after we discussed AWS services, Amazon Web Services, let's now look at another Cloud provider. In the end of a lot of these Cloud providers, I'm going to talk about some of the things that are common across different Cloud providers. So another Cloud provider is Google Cloud Platform. Google Cloud Platform, GCP otherwise is offered by Google. It's basically a suite of Cloud computing services that runs on the same infrastructure that Google uses internally for its own end products where this is like Google Search or G-mail and YouTube. You can see the same kind of generation of how these cloud platforms are being started. Amazon started the Cloud platform because of the spare capacity during off-peak hours. Google starts its own Cloud oftenly because it has such a great infrastructure and the background, for having very fast search and then supporting great infrastructure for offering e-mail services or even like YouTube services. So when you think about, for example, using Google storage, you can think about the same storage that Google Cloud uses, is the same one that they store their own assets as they're getting published on YouTube. Then, Google Platform provides the same infrastructure as a service, platform as a service, and serverless computing environments. We're going to discuss later on about serverless, and what does it mean really to use serverless. So when we see about the Google Cloud geographical map, you can see there's a bunch of data centers deployed around the world. From the United States, to Europe, to Asia, to Australia, to South America, we don't see actually one in Africa, and you can see that there are still continuously building new data centers. One of the difference for example, is Google has a data center in Los Angeles, AWS does not have one right now. So however, we see a lot of similarities, they have one in North Virginia, one in Oregon and so forth. AWS has one in Island, Google partly has one in London. So anyway, both of light is that they are distributing and they're building data centers all over the globe in order for people that want to access local data set, thus they can do that with effectively low-latency. So when you go to Google Cloud Platform, you get to these kind of website, where you got to use different types of service that they offer. The service they offer can be a batch, they have six other service or so, maybe a little less than what AWS provides. So they have an App Engine, Pub/Sub system, a Transfer Appliance, a Cloud Transfer Service, a Big Query Transfer Service, Cloud Functions. This is how we just upstream data. They have different things that store data, whether this is structured data, a database or big data, like Bigtable, Cloud Storage for a lot of data. Cloud SQL, if you're thinking of using different types of setups, says to do SQL queries to the database, memory store. If you want to use classes, don't mind the quiz if you do a database and different things like BigQuery to do big data analytics, to actually the most favorite shrine, which is Cloud spanner. So for those of you who don't know what's Cloud spanner, it's a very interesting paper. Highly I suggest reading that, I think have both of that in both of them in the course. [BACKGROUND] Another different services for process and analyze many of the big data sites under explore and visualize. One of the things that actually Google has advantage is it provides great interactions in their faces by the Google ecosystem, like the G-mail ecosystem with Google Sheets or so forth and the cool Cloud. Some that actually don't see that much in AWS, because there is no equivalent of C-Suite in AWS. Another thing that actually Google is actually pretty cool, the Google Cloud has the some of the AI machine learning with types of flow big, a very big one right now. They have popularized that very well within the Cloud, that's what [inaudible]. [NOISE] So when we think about the market study, we have so many Cloud providers, it's hard to compare and contrast and figure out who is actually winning the market share you know, obviously right now, AWS is about 30, 30 plus percent of the market share. Google is actually having second or third behind the Microsoft, though maybe second in some cases. But the probability is that back to the services or some of these Cloud providers other than AWS, they don't really publish their own revenue streams. So Google Cloud generated 2.6 billion in revenue in 2019. It's about the growth of 53 percent per year. So if you think about it, that growth is huge in terms of how much money they make year after year in Google services. For 2019 as a whole, Google bought about $8.9 billion. It's not yet a trillion business itself, but it grows with a 53 percent of growth, you can see how fast it's going to grow, which is of course, less than what AWS generated in the fourth quarter. So, if you ask me, which [inaudible] obviously AWS has a substance.

>> So let's move now to the last Cloud provider, which is Microsoft Azure. So Microsoft Azure is formerly known as Windows Azure, is another Cloud computing service created by Microsoft for building, testing, deploying, and managing applications and services through Microsoft-managed data centers. This is similar to the stories that we have heard with AWS and Google. It provides a similar software service, platform service, and infrastructure service, and some most different programming languages, tools and frameworks, including Microsoft-specific wireless and third-party software systems, something that we've seen in other Cloud providers. They may support, in some of the offerings, Microsoft specific products, but the integration you have with the.NET framework that Microsoft has produced, it's much better within Azure than what it is with other Cloud providers. However, if you think about how many users are actually using.NET framework may not be that many. Azure was announced in October 2008. Started with codename "Project Red Dog", and it was released two years later, and renamed Microsoft Azure in 2014. So this is mostly about history. But the interesting part of this history is that we can see that the difference between Amazon and Microsoft is that Amazon started maybe six to eight years before Microsoft did, which actually sees and indicates the reason why they have larger market share. It probably AWS has a much more mature Cloud ecosystem that Microsoft has, but we're getting to a point where Microsoft is actually increasingly really fast. If we'd actually deploy them, you can see they have from government, the specific data centers as you can see, in Virginia, in Central, in Texas and so forth. Even non-governments as well, like generic ones where people can actually use. They actually follow a similar pattern geographically; in India, Southeast Asia, Australia, Japan, and Korea are similar as we've seen Cloud providers. In Europe as well, different areas within Europe. Azure is generally available about 54 regions around the globe. Microsoft has announced additional 12 regions. That is as of the dated October 2018. Those regions increase really fast, actually trying to give a data point it's really hard with this course as we move from semester to semester. But bottom line, as of October 2108, the data indicates we have about 54 regions. Azure geographically contains multiple regions whether it's North Europe, West Europe and so forth, where a location represents the city or area of the Azure regions. Each Azure region is paired with another region within the same geography which makes them a regional each pair. In this example, Amsterdam, Netherlands, which form a region. This is some of the specific to Microsoft. Not many other Cloud providers actually follow that. So this is really interesting. So we see that the types of Azure Clouds similar to software as a service, platform as a service, infrastructure as a service, but the interesting part is that Microsoft offers integration with .NETServices, LiveServices, SharePoint Services, and even Microsoft CRM Services. The other beautiful thing is that similar to what Google does with the Google platform in the often, it gets with Microsoft Live and big and HealthVault and Microsoft Advertising and so forth. So if we see some of the symbols they use, they use symbols left and right, but the reason I put this slide is because similar technology in terms of availability zones, availability sets, single VM, region pairs, of course, it's specific to Microsoft not to others, but the rest of the terminology seems to be some standardizing across different Cloud providers. So one of the differences between Cloud providers and Azure uses its own fabric, its own layer that an operating system, which is called Microsoft Azure. It's the fabric layer. It's a cluster hosted of Microsoft data centers that effectively manages computer storage resource of the computers that produces the resources to applications running on top of Azure. This is different from what we've seen. For example, with AWS, Xen Hypervisor, more like an open standard one. Microsoft has its own type of provisioning the resources. Microsoft Azure has been described as a Cloud layer on top of a number of Windows Server systems such as Windows Server 2008. A customized version of Hyper-V, known as the Microsoft Azure Hypervisor to provide visualization of services. Again, you see different hypervisors. Again, AWS maybe using a different type. Actually, they use their own operating system to run their own services. Scaling and reliability are controlled by the Microsoft Azure Fabric Controller, which ensures the services and environment do not fail if one or more of the servers fails within the Microsoft data center. We've seen that in many Cloud providers, and which also provides the management of the user's Web applications such as memory and allocation and load balancing. Azure provides APIs, where we call it like REST APIs, HTTP ones, or even using XML ones. Microsoft provides a client-side managed class library that encapsulates the functions of interacting with the services. Again, this is common to all Cloud providers. They all provide libraries of how you can access the services. For example, if you go to access, the object store, they give you an object store as client that you can use inside the service itself to store large volumes of data. Now let's go to the benefits. One of the core strengths of Microsoft Azure is the ease of transition for organizations looking to migrate from different types of products like SharePoints to the Windows deployment. The task of administering certain technologies such as server Active Directory SharePoint can be really eased with combination of Azure and Office 365. Now the benefit of that is that it frees up IT staff to work on new projects rather than spending time on general system upkeep. So if you already are in the Microsoft ecosystem, moving to Azure it's easy. We see this idea of trying to integrate with some of the on-set hardware-based systems. With the CloudWatch, they make it easy to transition the customers from one to the other. Let's do a very simple overall

comparison. Amazon Web Services, like Amazon itself, aims to be everything to everyone. As such, AWS has the most extensive portfolio of Cloud services of any other Cloud provider. I said, one of the reasons is because they started early and much earlier than other Cloud providers. Google Cloud Platform's core strengths are in machine learning, big data tools, and extensive container support. A lot of that is because Google has had great experience. It actually was the first to [inaudible] a lot of the texts of all machine learning tools. So it makes sense for them to be ahead compared to others. At the same time, if you think about analyzing data, Google was bats ahead of collecting all this data than anybody else and become a data company. So that's why they have these extensive tools. So let us see like garden, this is an interesting report, how do we see the growth of the market? So this is billions of dollars. So if you think about this year big amount, $266 billion, we expected a few years to be 354 billions of dollars. So Cloud is definitely growing a lot. It has been growing a lot for many years. So if you actually see and you compare from 2018 to 2022, we see a doubling of the amount of dollars that will be generated in different areas of the Cloud. A batch of that is now this of the software services, not infrastructure which are commoditized, but actually all the software, when people developing software that runs on the cloud itself. So how big is really Azure? I mean, this is the big question that everybody ask. How big is the Cloud provider? There is still no publicly available data on Azure sales. Azure is the part of Microsoft's Cloud business that most rhymes with AWS, but it is buried inside the commercial Cloud. A commercial Cloud is a roll up of multiple services from Microsoft. So everybody is not likely to buy a buffet that includes Azure, but it is not focused on that. So Microsoft commercial Cloud is about 50 billion annual rate. So how do you compare that with other Clouds? It's hard. So while Amazon specifies AWS revenue, I think same to Google, it marks all reports as Azure's growth rate. So the number is 62 percent, which is a lot. If you compare that, like for example, 2018, it was about 76 percent last year, so it's quite a lot and pretty fast compared to what AWS grows, but as I said, AWS is much older and much more mature.

>> Let us talk about the Cloud Edge, that is the front layer of the Cloud. Clients can talk to Clouds using web browsers or the web services standards, but this only gets the outer skin of the Cloud data center, not the interior. So for example, considering on Amazon Web Services, it can host entire websites, for example, target.com, netflix.com, Walmart, and many others. Then it can also use data to actually store the objects like the files, the images, and videos within the website, and then the web service itself that is the server is hosted on top of an EC2 instance, or even it can provide virtual machines that you can do more things like more complicated logic. At the same time, the Cloud provides the ability to scale the service to handle increasing traffic through what we said before as the auto-scaling groups. So the high level, you see the users are trying to access the net and they usually hit the outer part of the Cloud. Client requests are handled by the first tier, and these can be like PHP or ASP pages. Then there's some logic that is being handled and this is what we call the frontend [inaudible] . [NOISE] The frontend developers [inaudible] [NOISE] These are lightweight services and very nimble, and many of these actually use caching. The reason we use caching is because you don't what for every request that comes to hit the [inaudible] [NOISE] information about a specific product that is associated with an ID. We don't want to always look for a product within the database [inaudible] you want to use a cache that you can optimize the queries being passed. In many cases, those caches are usually in memory. So you can really get past performance that have [inaudible] The way we have to achieve scalability is by using shards. So then you effectively use different shards of data that so you can horizontally scale the graphics across all the caches. So let us assume that you have about 1,000 keys that they represent specific approach objects in Amazon, like specific products. Then actually, what you do is you allocate a specific range of the keys within one shard, a specific range of the keys with another shard. So whenever you need, for example, to take a specific product, like buying from Whole Foods, some oranges, then you go to one shard, if you go to buy strawberries, you may go to the other shard. So then what you do is you load balance between different layers in the caching. [NOISE] So many styles of systems near the edge of the Cloud, when you focus on vast numbers of clients and rapid response, you want to really return back the response pretty fast because you know people want to have a great experience when they access websites. Inside, we find high volume service that operate in pipeline manner and they can operate as asynchronously. The reason they operate asynchronously is because if one of the services has an issue, for example, it has a congestion in the CPU or the network, you don't want to block the rest of the services, and let me give an example. When you go to amazon.com and you download the website, what happens is you actually have about 10,000 services that work simultaneously to give you this experience. One service may be the recommendation engine, the other service may be the cart, and so forth. So deep inside the Cloud, we see a world virtual computer clusters that are scheduled to share sources, on which public applications like MapReduce are very popular, and this is effectively the clusters that do what we know in big data. They analyze the data and effectively they provide recommendations back to the users, and the users of these are deep in the Cloud than being in the front layer, and these are done asynchronously. So when you access a website, let's say you go to netflix.com, you can select to see all the catalog. When you see the catalog, the whole machine learning happens. The backend that has [inaudible] data clusters to provide you the recommended movies for you to see. When we click a movie, the latency of actually getting it must be really small. So that's why these have to go to specific caches that sit, as we said in the previous slide, in the front, in number two here. [NOISE] In the outer tiers, replication is a key. So you want to have as many replicas as possible because for example, if everybody is asking for oranges from Whole Foods, especially during a crisis, that's where you effectively may be bottling. So you want to have 10 times those oranges or 10 times the key that represent the oranges between different data shards so that you can be able now to serve the tasks really fast. So we need to replicate this processing. Each client has what seems to be a private dedicated service for a little while. So you make a request, you buy the oranges. If somebody else doesn't request for an orange, then he may go to a different server on the same as you did so you don't get bottleneck. At data, you need to shard the replicated data as much as possible to have copies because you don't have delay. At the same time, you want to replicate the data because one of the instances goes down, and it happens a lot in the Cloud. We want to make sure your request does not fail, but rather you will be able to reroute that request to another node that has the data. So replication is also important for availability and provide proper service level agreements back to the users. In terms of control information, the entire structures managing an agreed [inaudible] [NOISE] by a decentralized Cloud managed infrastructure. Again, the notion of decentralization means that you're able to have multiple replicas. You're able to effectively shard the load so you don't bottleneck a specific instance, and that's why replication and sharding are the two most important elements when we discuss about distributed systems and we discuss about Cloud computing. You're able to achieve scalability because now you can scale the services, as well as being able to have replicas across these scalable services.

>> So let's talk about sharding. We briefly discussed before about the front layers and the edge of the Cloud. What about the shards? These are the caching components running in the tier-two, are central to the responsiveness of tier-one services. The basic idea is to always use cached data if at all possible, so the inner services, here the database or a search index stored in a set of files, are shielded from online load, we mentioned before of how we could have different caching technologies to do that. Many famous cache technology include Memcached or Redis, and many of the reverse indexing systems that we use to search, most common is Elasticsearch. So I can give you an example, when you go to Amazon.com and you search for a product, that's where you actually use the search index store to effectively identify what you're looking for. Then you return back replication, the key of the specific element that you're looking, for example this may be a dryer that you request from the database, and then from a database you get a list of all the images that you then request from the object store to get the images for those dryers. So you try to use caching as much as possible to effectively shield your databases from the load, because caches are usually run memory compared to database that need to store the data in a more durable store, like the hard disk or even an Elastic Block Storage if you use AWS or other services. We need to replicate data within our cache to spread loads and provide fault-tolerance. Again, we've mentioned about that, how for example if one of the instances that run the caching load goes down, how we should effectively have multiple replicas of the data so we can failover to another instance. In the background, once the new instance comes back with the same software, it's actually empty because it's a cache, so you lost the data in the memory, so you need to find ways to warm-up the data from the other replica. But not everything needs to be replicated, hence we often use shards with just a few replicas. So what is sharding? We touched it a bit more earlier on, but then we didn't really define what it is. So sharding is the process of storing data across multiple machines. So as the size of the data increases, a single machine may not be sufficient to store the data, nor provide an acceptable read and write throughput. So think about it. If you only have a single instance and you want to send thousands of queries per second, the network of that instance may not be able to handle it. If you have like one gigabit per second network or even 10 gigabit per second or 100 gigabit per second, still you may get a lot of load in a single instance. So what you need is you need to spread the load across multiple instances, and be able to use sharding technology to do that. With sharding, you add more machines to support data growth and the demands of read and write operations. So how do you do that? So since the tables are divided and distributed into multiple servers, the total number of rows in each table in each database is reduced. So think about it. Instead of having a table that has 10,000 rows, now if you have let's say four, you can have effectively 2,500 leads of the instances and you reduce that. So now what happens is the load per shard decreases substantially by effectively one-forth. So if you now escalate that to services like Amazon, Netflix, Facebook, Google, that tend to get like millions and billions of queries per second, then having shards is very important in order to be able to do that. Then on top of shards, you need replicas of the data systems themselves or the caches, such that you can achieve high availability at failover. So by sharding you reduce the index size of each of the instances, which generally improves the search performance as well. So you need to also shard your search indexes because if everybody is doing searches, not everybody is going to search the same item all the time, so that's important to shard the logic. But at the same time even if you have what we call hotkeys or hot shards, you need to find clever ways that you can effectively distribute the load in multiple systems. A database shard can be placed on separate hardware, and multiple shards can be placed on multiple machines, greatly improving the performance. Now, as we move from virtual machines to containers, now you may actually have containers in the same machine or different machines and you can also move containers, but we'll discuss containers later on. So think about the virtual machines on separate hardware, you need to be able to distribute the load across those virtual machines. So let's go and get an example here. The second tiers could be any of a number of caching services, I said Memcached/Redis are common technology, and I highly advise you to google these services and look at them, they're actually key-value stores. In memory key-value stores, they have what we call a distributed hash table that runs across all the shards, and then you use a key-value API. So for example, you want to search a specific item like a balloon, that balloon has an ID, let's say ID 100. Then it goes to that distributed hash table that says, "Okay, I have a 100, which node of all the shards is going to help me out?" So in this example, for example you have, this is the shard where you effectively split one terabyte of data in different shards. Another great across is Dynamo, Dynamo is a service created by Amazon, a scalable way to represent shopping cart and similar data. This was DynamoDB, Dynamo is the paper that actually was just created, DynamoDB has changed in the last like almost 30 years since the paper was written, so this is the database that Amazon uses to store all the data. Google uses Bigtable, it's another very elaborate key-value store created by Google, it's built on GFS. Then IBM has WebSphere eXtreme Scale though not so popular as the other ones. But effectively, if you want to read more, I have in the advanced paper at the end of the course, all these technologies like Memcached, Dynamo, and Bigtable, I highly advise you to look at them. It would be very useful to understand what is a distributed hash table, and how we can use these sharding

technologies. Then also sharding is cross-cutting, most of the systems replicate data to some degree as well. So as I said, you can't achieve everything with just sharding, you need to also use replication, they have to come together. You need to have multiple replicas of data and shard the data at the same time. So the question is; do we always need to shard data? So imagine a tier-one service running 100,000 nodes, can it ever make sense to replicate data on the entire dataset? Yes, if some information might be so valuable that almost every external request touches it. So if you have what we call a hotkey, you must think hard about patterns of data, access, and use. For example, you may have a unique and specific crisis, everybody wants to use toilet paper or a specific brand of a toilet paper. You are now able to have replicas and shards, because sharding won't work. Sharding effectively splits the datasets, as we said before in four different pieces, but the same key will be made on the same shards. So if you have replicas, now you can spread out the load in multiple instances. This is a good example in the case of a beehive. If we don't make a dynamic decision about level of replication required, the principle is similar. So if you have a beehive that increase, increase, increases, at some point the branch may break if you have it in only one dimension. But if you have across all dimensions, you may be able to spread the load and the branch may sustain. So this is the same idea in the Cloud, as I said in a crisis everybody goes for a toilet paper, so what happens is you better replicate the data across different systems, because sharding will not help, because in sharding you effectively break the keys, the key ranges in four pieces here. So if the toilet paper that you need is in shard one, that means that this will be overloaded and not much load will go to the other shards, so being able to replicate and shard together is where there's the gold dimension. So let's see about queries. So what happens if you try to do updates? So for example you have now three replicas, and you try to update one of them. For example, let's say the count of how many toilet papers you have bought. So let's say you update from, let's say one million to like one million and two or one million and one. How now are you able to make sure that when you make an update in one of the instances, in one of the replicas, that this information will actually propagate into other replicas as well? So you should also think about patterns that arise when doing reads. Queries are easy, queries is once you do an update, for example if all this is now one million and one and the load is like one million, and you try to a read, what you get is you may get incorrect information from one of the instances if the data have not been properly replicated. So effectively what we need to do is we need to do what we call a transaction, where you increase the count by one in one of the systems, and you make sure that you increase the count by one to the other one. So you have a coding system and you have the data, and that's very important when you think about updates.

>> So we have covered a lot of items in regards to sorting, in regards to cast casing, in regards to replication. Let's discuss about the critical path. [NOISE] The critical path is effectively a notion of what is important when a query is made or service so that it sorts properly. So what does critical path means? Let's focus on the delay until a client receives a reply, and that delay must be really small. The reason is because when you go and browse websites, you don't want to get responses slower than a few milliseconds. Just imagine, the blink of the eye is 400 milliseconds. So you need to get back responses and information really faster as a user. So it feels natural to browse across websites and feed the website. So critical path are actions that contribute to these delay. So let's assume now we get update this monitoring in the last criteria for Mrs. Marsh as follows. You get to ask for something on a service instance, on a Cloud, and then you have a delay before you say that. For example, you need to feeds all the data to reconstruct a website. An example is when you go and open your Facebook account, you want the news feed algorithm to work properly, you want the image to appear instantly, you want your messages and the list of the people to appear properly, and then the list of the groups and so forth. So the amount of time it takes from the click to login up to the time you actually get the reports back with all that information on Facebook, that's effectively this service a response delay. Then it's not only just how much the service is going to take, but it's also the network propagation, because when you make a request, you have to go to a server that runs by Facebook that may be a little distant. So that's why you see all these Cloud providers providing regions very close to where the users are or distribute across the globe with six regions and more than that, because they want to be close to that, so they want to decrease the delay in terms of the network. Assume now you're actually going in and updating your shopping cart numbers. You just added a new broccoli that you want to check it out. So if the updates are done asynchronously, we might not experience much delay of the critical path. For example, if you add a broccoli and you add an orange, and then once you to go the checkout, maybe broccoli is not available at the whole [inaudible] store, so Amazon will their back hold, we can find that oranges maybe available. If you actually block the request for all the things you have in the shopping cart, that may slow down, but if you're able to parallelise and do it asynchronously, you may have information in the shopping cart much faster. So updates are done asynchronously, we might not experienced much delay on the critical path. Cloud systems often need to work this way. So we need to avoid waiting for slow services to process the updates, but we may force the tier-1 service to guess the outcome. This maybe the case. For example, in Netflix, when you go and click that you want to see the catalog of the movies when you login, if the machine learning algorithm that runs and provides the recommendation does not work properly, and therefore, you have a slow response, what you do is you probably would fail back to a default response, the default list of the list of movies. So that's the thing, when you don't have proper response, you need sometimes to guess the response, even if this is your audience, and that's fine. Another example is when we go to the shopping cart, you have, let's say five broccolis, and then on the checkout, you actually see three broccoli, say, what the heck? There was an error. So that sometimes means that there was an error in the background that caused this, but then as a user, you can fix that. So sometimes inconsistencies that happened within the website, you propagate back to the user to fix. This avoids the slow service and whenever there's a problem, you actually can let the users know how to fix that. For example, you could optimistically apply update to value from a cache and just update this with the right answer. Many Cloud systems use these tricks to speed up response time. Parallelism is vital for speeding up first-tier service. When you, for example, go to website, you don't only want open one connection to save the whole website. You probably open like hundreds of these connections and therefore, your first-tier services have to be able to accommodate, not just 100 connections, but 100 multiplied by the customers who are accessing them. So parallelism is very important. That's why you you don't have a lot of threads, you need to have a series of the tier-1 because you want to have this parallelism across how you serve the request. So the request has reached some surface instance X. Will it be faster for X to just compute the response or for X to subdivide the work by asking sub-services to do the work? This is where you would go back to starting where effectively you make a request on other service to get that response. For example, you want to find out how many movies does Brad Pitt play. Then you want to have a contract service where you have the contract of the Brad Pitt. So what you do is you ask the service, give me the contract for Brad Pitt. Then you want to ask the service about how many movies has he played. So it's better to divide the work across the different instances of the services to be able to load, to search faster, and to find this formation united. Werner Vogels, a very famous person, he's the CTO of AWS, commented that many [inaudible] pages have content for 50 or more power subservices that are add a real-time audio requests. I can tell you that this was probably a like an outdated quote. Right now, when you try to access a website like Netflix, you have like 5-500 services that run in the background to serve that request. Not a single request, but overall. So 50 for a single client without pre-compute stat backup may be okay even today. So let's see now more examples, specific example. You go to the first-tier, you ask for a specific thing, and then you have this service response delay. The service response delay is comprised of all the latencies, of all the services that will help serve this request. As I said,

when you go to amazon.com, you see the shopping cart, you see the in-purchase, you see the recommendation for other services, you see the related products, you see the ratings. All these are different services and all the services have to give you a single website. So if the rating website is very very slow, you need to guess the rating in some cases. Therefore, the service response delay is the cumulative delay across all of these services. So this is the critical path, this is the critical path, this is the critical path and you can see, you parallel some of the request effectively, you can decrease the critical path delay. Now, let's go to another thing, what if we are unable to load balance between effectively some of the services? For example, we don't have a service instance per service, but we have hundreds or thousands per instance per service. Now we can effectively load balance out of that and be able to decrease the services response delay. So as you can see in this case, the most complicated actually is when you do updates. Updates are the most difficult because you update an item. Therefore, what has to happen is, you have to commit to that message to change and then the message has to be propagated back to the initial service. So for example, in this case, response delay usually would also include either latencies not measured in our work now the delay associated with the waiting for the multicasts to finish could impact the critical path even in a single service. So if you have like one service and you send a request for an update, you have to wait for everything to return back and then the cumulative delay is effectively the response from all of them. So you cannot really guess anything because these are updates and have to happen. So now, assume that you send an update, you want update the key, let's say the shopping cart, but you don't want to wait because all that rest of the saves are [inaudible] You load the website, you load the shopping cart, but for some reason, some has not loaded properly in your Amazon.com. So at the same time, you don't want to delay a user from checking out. You want to check him out so you can scout the other users. So the question may happen, several issues just can arise. Are all the replicas applying updates in the same order? What happens if you update, let's say for example, broccoli from 2-3, and then you update and all of a sudden, somebody else also buys a broccoli and now you don't have broccolis? So how can we effectively keep the order that the customer who ordered first is actually getting that and checks out, and then the other customer receives a consistent view that, it doesn't have to go like three websites on their own to see, now Amazon doesn't have broccolis for me. So are all the replicas applying updates in the same order? It might not matter unless the same data item is being changed. But then clearly we do need to have some agreement in terms of the order. What if the leader replies to the end user, but then crashes it and some of other updates were lost in the network? So you may update some. You may say, I won't update my broccoli. I want to have five broccoli, but then on the response, the service got that and updated your shopping cart, but then once you get back the response, the packets on the network were lost because there was some interruption. Then what you see is, you still don't see your broccoli is updated your shopping cart. This may happen. So we have to accept that networks can failed, datacenters can fail, or even things can happen that things may queue up. Still, you may have received response, but they queue up have in your network, so you haven't really seen that website yet. All these things is what would create what we call inconsistencies. Inconsistency is a big problem. When we discuss about Cloud computing, we'll discuss about distributed systems.

>> So welcome. Today we'll discuss the CAP theorem. We're going to start covering in these sections a little more about distributed systems and some of the basics on distributed systems. The reason we're going to do that is because a lot of the Cloud infrastructures we know today is actually based on distributed systems. So the fundamentals of distributed systems are super important when you discuss about the Cloud. In summary, the sections we covered about sorting in availability and reliability, those principles is how you design systems, on how you deploy them on the Cloud. So Eric Brewer is a famous scientist from Berkeley, he developed the CAP theorem. In his famous 2000 keynote at a conference called ACM PODC, he proposed that you can just do two from consistency, availability, and partition tolerance. Bear in mind, when you go especially to interview with big companies and to working in a Cloud business this is the baseline question they may ask, "What is the CAP theorem?" Effectively think about a small database and a database where you want to deploy this database that you want to store some data, and you want to deploy that on the Cloud. There are some principles that this database can have. These principles can be either consistency and availability, or availability and partition tolerance, or the consistency and partition tolerance. You can only pick two, you can't have all three. So consistency means that all nodes see the same data at the same time. So for example, if you have, as we discussed before, a sorted database where you house or if you have multiple replicas of the same data for availability reasons and you try to see the data, you have to see the data that are the same across two different nodes. This is really hard and the reason is because if you try to update the date in one node, the question is, how are these data being replicated to other node so you can get the same view of the data in the other node? Availability is a guarantee that every request that receives a response about whether it was successful or failed. [NOISE] As long as a response can come back, that is important and that notions of availability. But if the node is not accessible because it is down, then obviously this node is not available. Partition tolerance means the system continues to operate despite an arbitrary message loss or failure of part of the system. So let's think about partition tolerance. Partition tolerance is more focus, more about assuming that you have three replicas of the same data, like node 1, node 2, node 3. Then there is a network interruption between one of the nodes that is not accessible. What partition tolerance means that even if you do have this arbitrary message loss, the system can still continue to operate, and that's really important. We can see this in the triangle. I have a few of the most famous data systems here. In order of, where do they really fit in this triangle. For example, an RDBMS database like MySQL and Postgres or Aurora in AWS [NOISE] based on consistent availability. Whereas some others like CockroachDB, Cassandra, [inaudible] , DynamoDB, and CouchDB probably like a pretty old one, and Riak is also dead as well. R:Base on an AP system availability and partition-tolerance. Some other database like Mongo, HBase or HybridTable, or Bigtable, RCP Systems. When we discuss about interactions with web services, we have think about operations commit or fail in their entirety. So this has to happen. You cannot have, for example, "Hey, I just added this item in my shopping cart and half of it is there, but half of it is not there." This cannot happen. This is one we define atomicity, which means that the operation commit or either failed in its entirety. The second aspect is consistent. When you make a transaction and this transaction must be visible to all future transactions. So for example, let's assume that you created [inaudible] updates some keys in the database, and then you try to read those keys, you don't want to mentioned that the second transaction, it will see the data after they have been updated. The third demo we were going to discuss about uncommitted transactions are isolated from each other. So for example, you start transaction, then say, "I want to commit these three keys." But you say started transaction, you can meet the three keys, but you have any piece of transaction and this is uncommitted. So this has to be isolated from all the other transaction you can do. The last is durability. Once something has been committed, it's permanent. In the data systems, especially the ones who deployed the Cloud. When you have a transaction has been committed, that means the data have been flask down to the drive, so it is durable, that is permanent. So the most natural way of formalizing the idea of consistent service is an atomic data object. So definition is, there must exist a total order on all operations, such that it's operation looks as if it were complete at a single instant. This is equivalent to requiring requests, if the distributed shared memory to act as if they were existing in a single node, responding to all operations at a time. So what we go now is to go more deeper into the atomicity and the consistency. So atomic read and writes stored memory, any read operations that begins after the write operation completes must return the same value, or the result of a later write operation. What this means is that, whenever you do a write after read, we must make sure that what we read from the data system is the data that is the most recent after the write. As you used the CAP theorem, consistency is about two things. First, updates at the same data item are applied in some agreed-upon order. So as we go deeper and deeper in Cloud and distribute systems, we see that ordering is one of the hardest problem you can achieve. But what this means is, if you do update one, update two, update three, and update four, you want to make sure that your data system, when you read the data, they will showcase the same ordering. Second, at that once an update is acknowledges to an external user, it won't be forgotten. What this means is that, you want to make sure that once you have something durable, it's

going to be there, it's not going to be lost. When you request for that, it's not going to be forgotten. Now, obviously, make sense if you think about single node system, where you have a single transaction, you finish your edited on the hard drive. But now as you have three replicas and four replicas, and maybe 20 replicas or whatever it is, how can you think about consistency when you have multiple load? That's like a super hard problem. In some cases, not systems actually need both properties. When we design a system, we have to understand why we're designing that system. Sometimes we may relax one of these actually elements. If we do that, then we can design a more [inaudible] system, that can still have a market value and that can still serve a number of use cases. Let's go and see where the consistency is important. Because in some cases, consistency or inconsistency can cause substantial risks. So let's see. All of a sudden, you want to pay, you have a check that you want to effectively go to the bank and deposit it. All of a sudden you deposit a check, but then the data are being lost. So let's discuss about what happens when you deposit a check. You put a check, your Skype with your mobile phone, it gets to the banks. The bank, what it does, they open a transaction, they say, "Listen, this person has $10. Now he's going to have $20 because he just deposit a $10." The money cannot be lost. So when you discuss about banks or money, that's when you have to think about consistency. If we go back to what is important for the bank, it's important to be consistent and partition tolerance, CP system. They don't care about availability. What I mean by that, they don't care if the bank is down, fine, it's down. As long as the people were going to see a consistent view of the data after they post a check. They care about consistency and partition tolerance. Partition-tolerance means, for example, your dad access the bank from another state, they will be able to see the same data. So if you have a network partition between these two regions and the data not being propagated properly, all of us on the other area may not see the data. That's really bad. You want to make sure that whoever sees the data, can't see the data there. So weak or best effort consistency. Can we achieve something like, yeah, it's not going to be perfect, but let's assume there is a notion of best effort consistency. Strong security guarantees demand consistency, as we just said like security banks, money, these banks consist. But would you trust a medical electronic health record system or a bank that use weak consistency for better scalability? So what does weak consistency means? Personally, I will not trust. If all officer [inaudible] , the doctor believes that I have high blood pressure from what I told the database to have. So the question is probably no. You want to make sure that consistency is there. We discussed about atomic, consistent, isolated, and durable. Now this forms what we saw called an ACID. This is an acronym that many database developers use. Let's go a little deeper into that. So atomic, everything in a transaction succeeds or the entire transaction is rolled back, period. You do a transaction, you want to write some data in a database, if you don't finish, you have to rolled back. Then consistent, a transaction cannot leave the database in an inconsistent state. If you want to develop and you altered some data and your bank account. You cannot five dollars and then add the $10 later on. You have to make sure that the data is going to be isolated. Transaction cannot interfere with each other. How can this happen? Let's go back to the bank example. You have two people depositing money in the bank account. Two transactions happening at the same time. If those interfere with each other, all of a sudden you will see adding consistent view of how much money you have in the bank account. So that's why transactions cannot interfere with each other. Completed transactions persist, that's the durability even if where the servers restart. Believe me, servers can restart. Whether you call a bank, you call a Cloud provider or whatever it is, servers can dyke and restart, can do whatever it states. So if you have to have this ACID guaranteed in some very consistent system. This quality seems indispensable, and yet they are incompatible with availability and performance in very large systems. So going again back to the bank exam. We don't care about availability. Bank is not going to go down, we can deposit check a few minutes later. It's not a big deal. Of course, not many people deposit banks at millions per second. You see not a few of them in every hour or so, but you don't see millions per second. But when you discuss about online web services like Amazon, then you have millions of people accessing the service at the same time. Then consistency is probably not what you need. I'm going to go later on how you can achieve that. How inconsistency can be okay, actually.

>> So let's go and see now some examples in terms of the CAP theorem. So assume that you have a bookstore, that you shop around the corner. For example, suppose you run an online bookstore and you proudly display how many of each book you have in your inventory. Every time someone is in the process of buying a book, you lock part of the database until they finish so that all of the visitors around the world will see an accurate inventory numbers. That works well if you run the shop around the corner. But now let's assume that you have Amazon.com. [NOISE] Amazon might instead use some cached data because they have so many customers, and therefore they have to serve so many requests. Users would not see the inventory count at this second, but what it was say an hour ago when the last snapshot was taken. Also, Amazon might violate the "I" in the ACID, we just learned before, by tolerating a small probability that simultaneous transactions could interfere with each other. For example, two customers might both believe that they just purchased the last copy of a certain book. The company might risk having to apologize to one of them when they check out from their shopping cart, but that's fine. [NOISE] But that's actually much better than slowing down the website and irritating myriad of other customers. So what I mean by that is when you design systems, you may have systems, that can be okay in some cases the way you design them, and it can't be satisfactory even if you like some of these properties. So Vogels, again a famous CTO at Amazon.com. He was involved in building the new shopping cart service back like 15 years ago. The old one used strong consistency for replicated data, but the new and is built based on a Distributed Hash Table called Chord. It has weak consistency with eventual convergence. So this weakens guarantees, but the system is not delayed anymore, so you can get performance. You may get after the checkout that, "Hey, we cannot serve you this load." But still, you can push some of these problems after the customers in some cases, so you don't have to over design your system. So in this case, speed matters more than correctness because the user can himself fix the problem or get a notification at a later point in time. [NOISE]

>> So let us go a little more detail about the CAP theorem. So informally, let's discuss about eventual consistency. Eventual consistent informally guarantees that, if no new updates are made to a given data item, eventually all access to that item will return the last updated value. So we call it BASE (Basically Available, Soft state, Eventual consistent) semantics. We will see as we move forward how important it is to use eventual consistency. The core problem is, when can we safely sweep consistency under the rug? For example, if we weaken a property in a safety-critical system context [NOISE] , something bad can happen. But Amazon and eBay do well with weak guarantees between many applications just didn't need strong guarantees to start with. We discussed earlier as well, if you remember that there are some applications like banks that require strong consistency. But there are applications like the Amazon socket guard which can survive with weak or weaker consistency. So by embracing their weaker nature to reduce synchronization and so better response behavior. But what happens when a wave of high assurance application starts to transition to Cloud-based models where effectively eventual consistency is the case? So availability. For a distributed system to be continuously available, every request received by a non-failing node must result in a response. We discussed in the beginning about the CAP theorem. So any algorithm used by the service must eventually terminate. Therefore, it puts bounds on how long a service, an algorithm can run. But it puts no bounds on how long the algorithm may run before terminating, it therefore allows unbounded computations. In the case of partition tolerance, the network will be allowed to lose arbitrary many messages send from one node to another. That was the definition we gave earlier. But when a network is partitioned, all messages sent from nodes in one component of the partition to nodes in another can be lost. So the requirements is atomicity and availability. The atomicity requirement implies that every response will be atomic, even though arbitrary messages sent as part of the algorithm might not be delivered. The availability requirement implies that every node receiving a request from a client must respond, even though arbitrary messages that are sent may be lost. So the idea of partition tolerance, as I said at the beginning, is where, for example, you have three replicas of the same data and maybe one or even two replicas are partitioned from the other two because let's say the network is down between that or those nodes. But as we said before, partition-tolerance says that if the system can operate as if it was operating, if all nodes were available, then the system is called partition tolerant. So similarly expected to be highly available. So some of the services that we see the website is that we expect for them to be highly available. That means that even if for arbitrary termination of a virtual machine on the Cloud or arbitrary issue with a specific system, every request should generate a response. When a service goes down, it may well create significant real-world problems. The classic example of this is the legal difficulties should the E-Trade web site go down. Now the important thing here you have to abide is that we design system to be highly available so they can be up. But in some cases, if we say, hey, we don't care about availability, we care about consistency and partition tolerance, there might be legal implications to that. For example, if you cannot trade your stocks on a high day or on a low day, then you may have legal implications. So that's why when you design a system, you have to understand what really the customer use cases. However, when we design for example a shopping cart, we don't care about the consistency because we can push back some of the errors to the customers. So fault tolerance. When some nodes crash or some communication link fail, it is important that the service still perform as expected. One desirable fault tolerance property is the ability to survive a network partitioned into multiple components and we discussed it earlier what does network partition means.

>> So let us discuss now the most exciting finding, how does the CAP theorem correlate with Cloud computing? Is really CAP theorem valid in the Cloud? So the fact is data center networks don't normally experience network partitions, but when you think traffic between data centers, you may have what we call wide area link fails. Most services are designed to update in a single place and mirror read-only data across others. So the CAP scenario used in the proof can't arise. But the problem with this is when you design systems or when you have, I'm going to do updates in a single place and then I'm going to just replicate the data to some read-only partitions. The problem when you do that is, overall, the single place, a single machine, a single instance, and that cannot scale properly on the Cloud. So Brewer's argument about not waiting for a slow service to respond does make sense. It argues for using any single replica you can find, but this preclude that replica being consistent. So let's see some properties. Consistency; updates in an agreed order. We said that, durability; once accepted, it won't be forgotten. Real-time responsiveness, replies with bounded delay. Security; only permits authorized access by authenticate parties. Privacy; we won't disclose any personal data of the Cloud. Fault-tolerance; failures can't prevent the system from providing desires services. Coordination; actions won't interfere with one-another. [NOISE] So let's discuss about why did really others have develop, for example, Dynamo? Dynamo was developed with eventual consistency in mind because the strict consistency about writing a single note will not work properly in this case. So what we see more and more in the Cloud is that relational systems where you can effectively write data to update in a single note cannot scale properly after specific sites. So as we see the Cloud that we want to horizontally scale, the ability to have multiple access and be available and partition-tolerant becomes more and more important. Policy have been built with some of the properties they have. For example, Memcached does not provide special guarantees. Memcached is just a casid layer. So as long as you're able to access the service which is highly available, that's all you care about. Google GFS is file is current if locking is used. So it is a transactional system because it's a file system. BigTable is shared key-value store with many consistency properties and you can change that. Dynamo is the result technology for Amazon shopping cart and uses eventual consistency. Databases, you have different types of databases. Snapshot isolation with log-based mirroring. This is a fancy form of ACID guarantees. Map reduce; they use a functional computing model within which offers very strong guarantees. A Zookeeper is an interesting technology where effectively, it's not really a data system, but it's a system where you can have locks, you can use it as a coordination system. PNUTS is a Yahoo! database system. So it's data spectrum of consistency options. Great systems, great [inaudible] I don't think it's really used right now a lot. Chubby, similar to Zookeeper, is a logic service. It has very strong guarantees. So let's see if we get the conclusion. What thing to notes about the services. Most of the cost 10-100 millions to create. It's very expensive. Huge investment required to build strongly, consistent, and scalable high-performance solutions. Oracle; if you think about the ones that are actually right now are the most users in the Oracle data systems, it would require billions and billions of investment for many years to get it properly. CAP theorem is not about telling Oracle how to build a database system. CAP is according to you, that strong properties can easily lead to slower services. So if you use Oracle, you may get strong properties, but effectively, at some point, your master node or the one that gets all the updates will slow down. But thinking in terms of weak properties, it's often a successful strategy that gets a good solution and requires less effort. So that's the important thing. You can create the best system ever that has actual properties. But of the Cloud, I'm not sure this is what you need. It's probably the web services. This is not what you really need. At scale, at small scale, you can do whatever you like.

>> Let us discuss about consistency. I know we covered the CAP theorem, we covered a few of these elements. But consistency is still something that is a very interesting topic to go a little further into details especially when it is about distributed systems. Let's recall the cloud tiers, which actually many cases represents similar to what we see the tiers of the cake. Up to now our focus has been on the client systems and the network, and the way that the cloud has reshaped both. We looked very superficially at the tiered structure of the cloud itself, where tier 1 is very lightweight, responsive web page builders that can also route or handle Web Services method invocations. They're maybe limited to some soft state, so they don't really keep a state. In Tier 2 is where we have a key value stores and similar services that support tier 1. Basically, various forms of caches that can cache the data so the responses to the customer can be really fast. Then during the inner tiers, we have this online services that handle requests not handles in the first tier. These can store persistent files, run transactional services, but we shield all of them from the load. Effectively, the back end runs offline services that do things like indexing the web overnight for use by tomorrow morning's tier 1 services. Effectively, what we see is you have like tier 1, tier 2, inner tiers and then you also have the back end system. Central feature of the cloud is of course the replication. The reason you do that is because you need to handle more work. In order to do that, you need to make more copies. In the first tier, which is highly elastic data center management layer, pre-positions inactive copies of virtual machines for the services we might run. Exactly like installing a program on some machine. If load increases, we create more instances. Effectively, we create more nodes over the instances themselves. Then we use a load balancer that will spray the requests across the nodes. If load drops, so that means that we just kill the unwanted copies. We don't need any really warning for that. The elasticity of the cloud allows through by having replicas of the system itself. Things we can replicate that a Cloud, file or other forms of data used to handle requests. If all our first tier systems replicate the data needed for end-user requests, then they can handle all the work. There are two cases to consider, in one the data itself is write once like a photo. Either you have a replica or you don't. In the other the data evolves over time, like the current inventory count for the latest Ipad in the Apple Store. We can also replicate computations. Here we replicate some request and then the work of computing the answer can be spread across multiple programs in the cloud, some a scatter-gather way. We benefit from parallelism by getting a faster answer because now we can spread the requests across different services. At the same time, this can provide some form of a fault-tolerance, so if one service fails, you still are able to respond back with the query, a based on what you have received from other services. Many things map to replication as we just saw data or databases computation, fault-tolerant request processing, coordination and synchronization like who's in charge of the air traffic control sector over Paris, parameters and configuration data, security keys and list of possible users and the rules for who is permitted to do what, and membership information in a DHT or some other service that has many participants. All these effectively got it so far, get replicated so that we increase parallelism and increase the performance of the system of the services. But replication then becomes the issue of consistency. We would say that a replicated entity behave in a consistent manner if it mimics the behavior of a non-replicated entity, and this is actually really hard. For example, if I ask it some question, and it answers and then you ask if that question, your answer is either the same or reflects some update to the underlying state. Many copies but acts like just one, so what you have with replication is you have all these copies across the board. But if you ask the same question twice, you need to get the same response back. An inconsistent service is one that seems to be broken like you ask for a request, you get less [inaudible] but you are who you ask I requested it would be you actually get something different and that inconsistency is what appears to be as a broken system. A consistent distributed system will often have many components, but users observe behavior indistinguishable from that of a single-component reference system. You have a reference system and then you have an implementation. As you can see, looking at the clock itself, only it appears [inaudible] because it says the date and the time. But it's about more complicated inside that. The dangers of inconsistency is that it can cause bugs. Clients would never be able to trust servers. The problem with these bugs is that because it's inconsistency, bugs appear at random times without the need to know what actually really caused the problem. Weak or best effort consistency which is common in most of today's replication systems. But strong security guarantees demand consistency. The question is, would you trust a medical electronic-health records or a bank that used weak consistency for better scalability? I think the answer here is very interesting. Sometimes you can, sometimes you cannot. Let's say you rent a check and then it bounced back, that unit had insufficient funds. You know, that's a major problem.

>> ECE 530 Cloud Computing. Today we're going to discuss about distributed systems. I will say that a lot of the principles we have discussed about Cloud computing is correlated with distributed systems. Today we're going to go a little more in the definition of the correlation between distributed systems and Cloud computing. [NOISE] If we start first of all, defining what is a distributed system, based on the free online dictionary of computing, called FOLDOC. A collection of probably heterogeneous automata whose distribution is transparent to the user so that the system operates as one local machine. This is in contrast to a network where the users are aware that there are several machines, and their location, storage replication, load balancing, and functionality is not transparent. Distributed systems usually use some kind of client-server organization. The question here is Cloud computing and distributed systems based on this definition? The answer is yes, because the customer does not see all the functionality on the machines or the backgrounds, but assumes that this is taken care by a vendor that provides this as a service. Let's have a look at a few more definitions, based on the distributed system is a collection of independent computers that appear to the users of the system as a single computer. With Michael Schroeder, the definition is; distributed systems is several computers doing something together. The distributed system has three primary characteristics: multiple computers, interconnections, and shared state. As we see across the three definitions, what is the distributed system, is something that appears as a single element to the user regardless of the complexity of that. When we try to deal a little more about the problems that we have seen in distributed systems, or that we will observe failure detections, time and synchronization, global state and Snapshots, multicasts communications, mutual exclusion, leader election. All of these are basic theoretical concepts. If we had the time, a distributed systems course will actually focus in all of these elements. However, our focus is on Cloud computing, and a few of the broader aspects is Cloud computing is that of peer-to-peer system. Though some of the elements of Napster, Gnutella, Chord, which is a pretty big systems, are mostly outdated. So Cloud computing, Hadoop, are more correct to what we're doing and what lies beneath those distributed systems is either a network and routing, is a sensor network, or Datacenter disaster case studies. If we see a few of the problems that exist, RPCs or Distributed Objects. These are the basic building blocks that we will see within the Cloud computing area, and distributed services like storage, where we did concurrent control or two-phase COBIT or three-phase COBIT, Paxos is a very famous protocol, and I believe we have the white paper at the end of the course. Replication protocols are very important to achieve, multiple replicas at high availability and gossiping. We will discuss about Cloud computing, some of the elements, key value, SQL stores, and we will cover those later on as well. Stream processing, like data in transit, is also important. There are also some problems of distributed systems that I would say is fairly old, or older Cloud computing like self-stabilization, distributed file systems or distribute shared memory. These are fairly similar distribution problems, but all the steps of the timeline. What is important is when we deal with systems, whether it's Cloud computing or not, is to think about security and Byzantine Fault Tolerance. We will discuss later on in detail about the Byzantine Fault Tolerance. The Distributed System Design Goal is to achieve heterogeneity, robustness, availability, transparency, concurrency, efficiency, scalability, security, and openness, and we'll discuss a few things about consistency, cap theory, partition-tolerance, Acid, Base, and many, many others. Let's go into the definition of these goals. Heterogeneity is different types of servers, of networks, of applications, of services, of consistency guarantees. Now we have different types of many things, whether it's a network, an application of service, consistency guarantees, whatever it is. Robustness is fault-tolerance to a variety of failures. We need to make sure that our system appears as big, robust, regardless of how many services, microservices, or even services that are being used at the background, the system has to be fault-tolerant. Availability of data, of operations, in spite of random failures and network partitions, it is very important. You need to have a notion that this is available for this amount of time. Transparency: provide an abstraction of one property while allowing sufficiently flexibility at run-time, like Clouds, transactions, synchrony, sequential consistency and so forth. Concurrency: support many clients. When you think about Cloud computing, the reason you moved to the Cloud is because you wanted to scale your number of customers or your infrastructure to do that. So concurrency is important is important as an element of many nodes that need a lot of concurrent access. Efficiency: now you need fast, your critical path must be as low as possible in terms of time. If it is really high, then it won't serve the needs of the application itself, and you need to scale your servers to accept a lot of throughput, not only latency but through what's is important. Scalability: many operations per second, in spite of thousands of servers or millions of customers, so you have to scale. Security: system should be protected from attackers and bugs, like encryption and signatures, and is important, because once you give the data to the Cloud, you need to make sure that it is properly secured to achieve the goals. Openness: each service or protocol can be built or other services or protocols, it's layered stack architecture. This is like understanding the system, how it is open, being open about that is very important. We'll discuss about the intricacies of distributors; synchronization, concurrency, failures, and consensus, are the main four properties. Synchronization mean that you need to have multiple clocks, and

effectively you need to have multiple nodes, and all of them have to agree to that clock. You will see that this is very, very difficult to achieve, all agree an exact time. Concurrency is when you have different concurrent access, but you need to make sure that if you do have to access, you don't have conflicts or you don't have a different response, based on the access itself. Failures: You have to handle probability of failures. The problem with distributed systems and Cloud computing, is that failures can happen in different spaces like a network partition or multiple simultaneous failures or whatever it is, and you need to make sure that your system is able to operate under any of these random failures. Consensus is difficult to reach consensus, for example, if you have multiple replicas of data, you have to have some sort of consensus, what is the correct information? This has to happen when there is a failure or when there's a lack of synchronization, and even more in [inaudible] What we're trying to do in this lecture is cover the surface of the problems and go really deep into each of them because it's very complicated. In terms of the models, we have a client-server approach, we have a multi-tiered approach, a peer-to-peer approach, an agent-based approach, multiple codes like Applets or mobile agents, service oriented computing, and then we also have Cloud computing. All of these effectively are approaches of a distributed system, or limitations of a distributed system. Then we have the Emerging Computing Model, like the Cloud, the Edge, and Beneath. It's cheaper to provide services to a thin client than it would be to maintain a fat client in a changing world. Effectively, fat clients means that you provide an installable to your machine. For example, when you want to play a game, you install a game on your laptop, or when you install, you install some sort of an application that has all the data. What it says here, is providing a feed client, like a web browser, but you just access something with it, that's the thin client perspective. As we move towards battery limited devices, like mobile phones, and IOT and Edge devices, [NOISE] having a thin client is important because now you remove a lot of the computation from the device back to the Cloud. Besides the Cloud, all other computing infrastructures become simply Edge Devices to the Cloud. The other question is; does IOT count in this new world of Clouds? Or are Sensors Edge Devices? When I'm thinking about that, the case of the Edge, you have effectively the Edge accesses the services in the Cloud. We discussed about Tier 1 and Tier 2. Then you have different types of sensors that do also access those Edge part of the network, and then the services, Tier 1 accesses the Tier 2 parts of the Clouds so effectively, in my opinion, there is this [inaudible] where we discuss about the Edge to Sensors versus where we discuss Cloud. You have Sensors, Edge and then you have the Cloud.

>> In the first part of this series of lectures around [NOISE] distribute systems, and replication, and availability. We discussed about being able to effectively have multiple replicas of the data. As we went over a little on the surface of some of the problems of distribute systems, the consistency between the replicas was a very hard problem to solve. Let's recall from the previous slides that Cloud-scale performance centers on replication, effectively have more applicants because you can have multithreaded access, you can have multiple machines setting the workload. Then consistency of replication depends on our ability to talk about notions of time. Let us use the terminology like If B accesses serves S after A, then B receives a response that is at least as current as the state on which A's response was based, right? That means that we have to keep some sequence of operations based on when it happened. One of the very famous scientists around distribute systems, his name is Leslie Lamport, said don't use clocks because when you have multiple replicas, one thing that is going to happen is that you may have what we call clock skewness. So usually in cloud computing, what happens is we have a centralized time machine. Then all these different replicas it try to synchronize with that time machine. But there are different failures that can happen with that for example, what if you have a network partition and all of a sudden, one of the instances cannot access the time. What Leslie Lamport says, instead of using time and the real clocks, let's use logical clocks. If A happens after B, sorry, if B happens after A, then that's a sequence of ordering. That's when we went to logical clocks. Then it went deeper into vector clocks. But we don't have to cover it in the details right now. We also explained the notion of an instant in time and related it to something like a consistent cut. The next step is, we'll create the second kind of building block for effectively keeping. Trying to figure out how can keep effective replicas in sync, and that's the two-phase commit. There's also a more complicated one which is called the three-phase commit. These commit protocols or any similar pattern arise often in distributed systems that replicate data as we just said. This is very common framework that happens, especially NoSQL databases. Closely tied to consensus or agreement on events and event order, and hence replication. The two-phase commit problem is the problem that first was encountered in database systems. Suppose a database system is updating some complicated data structures that include parts of residing on more than one machine. Effectively what you want is you have this huge data structure that exist across two machines and therefore you have to effectively take it from both. As they execute a transaction is built up in which participants join as they are contacted. So what's the problem? Suppose that the transaction is interrupted by a crash before it finishes. But it was initiated by a leader process which we call it L. By now, we've done some work at P and Q, but a crash causes P to reboot and forget the work L has started. This implicitly assumes that P might be keeping the pending working in memory rather than in a safe place like on disk. But this is actually very common to speed things up, right? You keep in memory, you don't write this. Forces writes to a disk very slow compared to in-memory logging of information, and persistent RAM is usually very costly. How can Q learn that it needs to back out? The problem here is we have a transaction initiated by L, some work done by P and Q. But a crash courses P to reboot and forget the work L has started. Effectively, the question is, how can we get back to the state and how Q learns that effectively it needs to back out because P has been rebooted. We make a rule that P and Q and other participants in it can extend to many instances, treat pending work as transient. You can safely crash and restart and discard it. If such a sequence occurs, we call it a forced abort. Assume that if P fails, then the whole thing aborts. Transactional systems often treat commit and abort as a special kind of keyword, so effectively it says I'm starting a transaction, then I'm committing that, and if nothing happens then I'm going to abort the transaction. Let's see, L executed transactions begin. You read some stuff, get some locks, do some updates in P Q, and R and finally good commit that. So if something goes wrong, you execute on a board. If something goes wrong, then abort message goes back to L. Begin has some kind of system aside id. Effectively what this means is you are assigning a transactional id. You know the information. It acquires pending state, updated it at various places it visited. Read and update or write locks as it acquired. If something goes horribly wrong, you can abort it. Otherwise, if all went well, we can request a commit, but even the commit itself can fail. This is where two-phase commit, and three-phase commits algorithms are effectively used. Effectively if during the transaction you fail, fine, you can abort it. But if everything is successful and then you commit, but then the commit fails, that's where the problem usually arises, right? That's where effectively you need some more complicated logic to handle the two-phase commit and three-phase commit. The two-phase commit problem, the leader L has set of places like PQ it visited. Each place may have some pending state for this, it takes form of pending updates or locks held. L asks, can you still commit? P and Q must reply, no. If something has caused them to discard the state of this transaction like last updates or even broken locks. Usually occurs if a member crashes and then restarts. No reply treated as no, like once you have a no, effectively it's a failed member. If it's a yes, if a member replies, yes. It moves to a phase where we call it prepared to commit. This is very important because we said in the whole transactional items, if anything fails, the whole operation fails. But if the commit fails is where the difficult part is. But if everything's good and we get the prepared to commit state. After when it could just abort and you tie the way, as we said, if data or locks were lost, you at the crash

or a timer. But once it says, "I've prepared commit." It must not lose locks or data. It will probably need to force data to disk at this stage. Effectively, you can still have performance by keeping data in memory, but once you get to the prepare to commit phase, that's when you need to write to the disk. Many systems push data to disk in backgrounds so all they need to do is update a single bit on disk. Is called prepared equal true. But this disk that writes is still considered a costly event. Then you can reply back. 'Yes" So L sends out, "Are you prepared?" It waits and eventually has replies from P and Q. "NO" if somebody replies no or if a timeout occurs. "Yes" if only that participant actually replies, yes and hence is now in the prepared to commit state. If all practice participants are prepared to commit, then L can send a commit message, else L must send an abort notice. Note that there could be a mistake in abort, and this is okay. Because even if it aborts [inaudible] it's just aborted it. But if everybody is prepared to commit state, you're in this phase, you can select commit message and then you are done. If a participant is prepared to commit, it waits for an outcome to be known. It learns that a leader decided to commit, it finalizes the state by making updates permanent. Unless the Leader decided to abort, it discards an update, and then effectively it can release the lock. As you can see, we went to the prepare to commit phase. We've got the commit, then finally everything has been finalized.

>> So let us discuss about some theoretical aspects of some of the failures and the Byzantine failures. So many things that a 3PC commit as a practical protocol in order to detect failures. But to really use 3PC commit, we will need a perfect failure detection service that never makes mistakes. It always says, P has failed if, in fact, P has failed, and it never says that P has failed if P is actually up. So the question that we have to answer is, is there a possibility to build such a failure service? So let's discuss about failure in models. At the Cloud, we have seen different types of failure models, and each of them can cause different reactions to the systems. So many things can fail in distributed systems. A network can drop packet, or even once the packets have arrived to the host, the operating system can do so before they get to the user space. Links can break causing a network partition that isolates one or more nodes and it can cause partition tolerance. Process can hang by halting suddenly. This is especially important in JVM, Java-based systems, where you may get these huge spikes in your JVM that you don't know really what happened, but may cause an issue. A clock into the system can malfunction causing different types of issues in terms of reporting in correct time. So a machine could freeze up for a while and then resume. Process can corrupt their memory and behave badly without actually crashing. A process could be taken over by a virus and might have a malicious way that deliberately disrupts a system. So if I get bad actors exists on the Cloud it can give you these attacks. So let's see, the best case scenario is fail-stop with trust notification, and the worst case scenario is having what we call a Byzantine failure. In 2008, Amazon S3 was brought down for several hours when a single-bit hardware error propagated through the system. So what this means that a single failure in a single subsystem could cause the whole thing to go down. So the question is, how do we protect against that? Linux and Windows use timers for failure detection. These can fire even if the remote side is healthy so we get inaccurate failure detections. Of course, many kinds of crashes can be sensed accurately, so for those we get trusted notifications. Some applications depend on TCP, but TCP itself uses timers and so has the same problem. So the bottom line is, when you start using timers to detect something and make a correlation, this happened after that, then subsystems we use timers which may also be reporting incorrectly. So then, how do you really figure out what was the cause of the issue? A Byzantine case, this is like the Byzantine model which is the Byzantine Empire back in days Greek before, after the Roman Empire, that was Byzantine Empire which was based with a Greek language. The idea is, since programs are buggy, it can be appealing to just use a Byzantine model. A bug gives a random corrupt behavior like a mild attack. What I forgot to say is, Byzantine failed at discourse the automatic byte will get resurrected. But going back to this slide, Byzantine model is hard to work with and it can be costly, you often must, outvote the bad process.

>> So let us see now some examples of failure modes that can happen on the Cloud. So external case two-phase commit and three-phase commit normally used in standard Linux and Windows systems with timers to detect a failure. Hence, we get inaccurate failure sensing with possible mistakes, P thinks L is faulty but L is fine. Three-phase commit is also blocking in this case, although less likely to block than two-phase commit, and it can prove that any commit protocol would have blocking states with inaccurate failure detection. So Werner Vogels: World-Wide Failure Sensing. Vogels wrote a paper in which he argued that we really could do much better. In the Cloud computing setting, the cloud management system often forces slow nodes to crash and restart, and uses as a all-around fixer-upper. Also helpful for elasticity and automated management. [NOISE] So this is true. What happens is many time actually, they believe that instead of actually waiting forever and causing your whole system to halt, especially when you're trying to do like three or two phase commits, it's better to actually kill the slow nodes and have abundant time of errors. So in the Cloud management layer is a fairly trustworthy partner, if we're to make use of it. AWS actually does that well, when they detect a potential failure node because a node is a little slower or whatever, they totally terminate it and they can take it out, and then they just bring a new hardware or a new VM. Suppose the mailman wants a signature. He rings and waits for a few seconds. Nobody comes to the door, should he assume that you have died? Hopefully not. Vogels suggests that there are many reasons a machine might timeout and yet not to be faulty, as we see in this example. Scheduling can be sluggish. So what can cause delay in the cloud? So scheduling can be sluggish, like when you schedule something, how long does it take to schedule and so forth? A node might get a burst of messages that overflow input sockets and trigger a message loss or a network could have some malfunction in its router or links. So that means that although the message can arrive or they can be lost, but we don't know when it will arrive because of input queues and message losses. A machine might become overloaded and slow because too many virtual machines were mapped on it. So this can be like the actual hardware can be overloaded. Even the JVM, honestly, within a virtual machine can halt. So for example, if you write your software in Java, Java uses its own virtual machine, which is called JVM. Actually, the benefit of that, it allows portability between operating systems at the same time. It's a virtual machine so it can have the last word. So an application might run wild and page heavily. Page may be like you may get or you may have memory leaks so it can cause taking a lot of memory. So Vogels suggests we recommend that we add some failure monitoring service as a standard network component, instead of relying on timeout, even protocols like RPC, TCP could ask the service and it would tell them. [NOISE] It could do a bit of sleuthing first, ask the operating system or that machine for information, and then check the network. So why clouds don't do this. In the cloud, our focus tends to be on keeping the majority of the system running, no matter what the excuse might be. If some node is slow, it makes more sense to move on. Keeping the cloud up as a whole, it is way more valuable than waiting for some slow node to catch up. So in the end of the day, what matters is the end-user experience. So what experience do you get when you use VMs? So the cloud is causal about killing things, it avoids services like failure sensing since they could become bottlenecks. So what bottom line is what I mentioned before, is in order to have a good experience, it's better to kill an instance if you think it's slow than actually keeping that and have big slow because it can cause other errors within services as well and probably disorders to other systems. Also, most software is buggy, so let's now go into details about two types of bugs; the bohrbugs and the heisenbugs. These based on Bohr and Heisen, two very famous scientists. Bohrbugs are boring and very easy to fix, like Bohr problem of the atom. You get to figure out, "Oh, there's a problem here. It needs a fix, I'm going to fix it and I'm going to move on." Heisenbugs are the most difficult one. They seek to hide when you try to pin them down, caused by, for example, concurrency and problems that corrupt data structures. It could be visited for awhile. But it's hard to fix because crash seems unrelated to the bug. So you try to say, "Oh, this is the reason why it crashed," but then you can't really figure out that bugged out, what was the actual bug that caused. Studies show that pretty much all programs retain bugs over their full lifetime and that's true. Actually, many software are being created over a duration of like let's say five years or so. So all of a sudden, we have a new version or new variation, you always have bugs that, in the beginning there will be a lot of bohrbugs and then you may have a lot of heisenbugs. So if something acting strange, it maybe actually failing. Worst of all, its timing can also be flakey. When you have like billions and billions of dollars that you scale, you cannot trust timers at all. Too many things can cause problems that manifest as timing faults or timeouts. At the same time, with millions of nodes and billions, it's hard to even synchronize the fibers within all these nodes. So for example, how do you get all that to work? If a node fails, that actually happened before another one if their timer is running faster. Again, there are some famous models and again, none is ideal for describing real clouds. Real clouds is what we said, it's a very difficult distributed systems problem. We have to absorb that, timing can be a significant issue in the cloud.

>> So let's move on from the failure modes in the baseline failures to synchronous and asynchronous execution. I know we're getting a little deeper on some of the theoretical aspects of distributed systems, but I think these are foundational aspects when we think about Cloud computing. So let's look at three processes, p, q, and r. In this case, something happens from p, you have something that happens from r to q. Some data, let's say, being sent. In the synchronous model, messages arrive on time. Processes can serve like a synchronized clock. This is the synchronized clocks. For example, if a failure happened here, we can figure back whatever happened. But in the asynchronous model, none of this really holds because you don't have a centralized clock to effectively synchronize everything. So the thing is real distributed systems are not synchronous. They're asynchronous. You may have a central clock you monitor, let's you may have like a clock that you monitor all the instances, but the interaction with this clock may fail. But I said they're not asynchronous. Software often treats them as asynchronous. In reality, clocks can work in some times. In practice, we often use time cautiously. It can put limits on message delays. But in reality, it doesn't mean that clock always works well, but works most of the times well. For our purposes, we usually start with asynchronous model and the reason we do that is because it's better to start something that you know that may happen than with something than one that you're not guarantee is going to happen 100 percent. Subsequently, we enrich it with sources of time when useful. So that's what I mean. It's actually asynchronous, but if I can use sometimes the timer in some cases, O can have that bounded with a specific time, then I can use it. We sometimes assume a public key system. This let's us sign in or encrypt data where need arises. So Jill and Sam will meet for lunch. They'll eat in the cafeteria, unless both are sure that the weather is good. Jill's cubicle is inside, Sam will send an e-mail. Both have lots of meetings, so might not read the e-mail, she will acknowledge his message. They will meet inside if one or the other is away from their desk and misses the e-mail. Sam sees sun. Sends e-mail. Jill acknowledges. Can they meet outside? Well, the questions is, this is a common problem when we send messages, are we guaranteed that they will meet outside? So let's see. Sam sees the sun. Jill, the weather is beautiful outside, let's meet at the sandwich stand outside. I can hardly wait. I haven't seen the sun in weeks. Jill sends an acknowledgment, but does not know if, for example, Sam has read it. If I didn't get her acknowledgment, I'll assume she didn't get my e-mail. In that case, I will go to the cafeteria and she's uncertain, so she will meet me there. So you send a message, you get it back, you acknowledge, yes, great. See you there. Jill got acknowledgment, but she realizes that Sam won't be sure she got it being unsure he's in the same state as before. He will go to cafeteria, being dull and logical, and so she meets him there. Now, Jill sends an acknowledgment. Sam acknowledges the Ack, Jill sends acknowledges the Ack of the Ack. So effectively, what this means is that if we're to have a code system protocol after many interactions, everybody would acknowledge the other person so that you have a guaranteed that it has been received, and this can go effectively forever. Going back to this example, yes, great. See you there, but then Jill has to acknowledge that she has received the message and acknowledge, and acknowledge, and acknowledge. So suppose that noon arrives and Jill has sent her 117th acknowledgments. Should she assume that lunch is outside in the sun or inside cafeteria? What may happen is [LAUGHTER] you may get to the point where it's probably too late for lunch, so maybe tomorrow. The other aspect is that even if you set the sequence of messages, you don't know if the other person actually saved it. So we can't detect failures in a trustworthy, consistent manner. We cannot reach a state of common knowledge concerning something not agreed upon in the first place. We cannot guarantee agreement on things in a way certain to tolerate failures. So this is the bottom line. When do you send the message in the process in the Cloud service systems, it's extremely hard to assume that you can tolerate all the possible failures. You have to acknowledge that it will be [inaudible]. So going back to two-phase cubit and third-phase cubit. [inaudible]. Three-phase cubit is definitely better than two-phase cubit. In the real-world, a three-phase committees mostly costly extra, but blocks just the same. It can have it accurately. Failure detection. Failure detection tools could generally help, but the Cloud thread is in the opposite direction. Cloud transnational standard requests are inactive here. [inaudible] service, if it goes down, the Cloud transaction subsystem halts until it restarts. So bottom line, many systems that we will be using to face [inaudible] , but those early to terminate transactions in a transactional system. So this is what we'll be using as the best effort system. [NOISE]

>> Let us discuss today about data partitioning. In the past, we have discussed about the DynamoDB data. One of the features for DynamoDB was the hash-based partitioning algorithm that it uses to distribute the data across different nodes. We will discuss a few of the partitioning methods, the possibilities of how you can partition some of the data. Let's define what is data partitioning. Data partitioning is a technique to break up a big database into many smaller parts. It is a process of splitting up a database or a table within a database across multiple machines to improve the manageability, performance, availability, and load balancing of an application. Effectively, if we have a database, we break it into smaller pieces, so every access to part of the database can represent partially the load, at the same time, if that node goes down, we still have replicas, but the same time we don't lose full availability of the cluster. The justification for data partitioning is that after a certain scale point, it is cheaper and more feasible to scale horizontally by adding more machines than to grow it vertically by adding beefier servers. Of course, this is one of the substance and privileges of Cloud computing. We discussed how we wanted to scale up systems in the Cloud horizontally versus vertical. Now there are many different schemes one could use to decide how to break up an application database into multiple smaller databases. Below, we're going to describe three of the most popular schemes which is used across various large-scale applications. One is horizontal partitioning, the second is vertical partitioning, and the last one is directory-based partitioning. In horizontal partitioning, we put different rows in different tables. Let's now see an example. If we're storing in different places in a table, we can decide that locations with ZIP codes less than 10,000 are stored in one table, and places with ZIP codes greater than 10,000 are stored in a separate table. This is also called range-based partitioning as we're storing different ranges of data in separate tables. Horizontal partitioning can also be referred to as data sharding, and of course, we have defined sharding earlier in this course. The key problem with this approach is that if the value whose range is used for this partition isn't chosen carefully, then the partitioning scheme will lead to unbalanced servers. An example is splitting locations based on their ZIP Codes assumes that places will be evenly distributed across the different ZIP codes. The assumption, of course, is not valid because we have thickly populated areas like Manhattan or Los Angeles, or Chicago. Then we have suburban cities, and we have also smaller areas where there is huge areas that cover ZIP Code with much less population. In this example, if we use horizontal partitioning, the problem is that the load in Manhattan, maybe higher than the load in other places. This is a problem because let's say within the COVID situation that we had a lot of sequences in Manhattan area, that if we had stored this data in another database to analyze that would have become what we call a hot site or hot partitions. Vertical partitioning is when we divide our data to store tables related to a specific feature in their own server. An example is if we're building Instagram-like application, well, we need to store data related to user photos, users upload and people users follow. We can decide to place user profile information on one database setup, like friends list on another, and photos on a third server. Vertical partitioning is straightforward to implement and has very low impact with replication. The main problem with this approach is that if our application experiences additional growth, then it may be necessary to further partition feature specific database across various servers. It will not be feasible for a single server to handle all the metadata queries for 10 billion photos by, let's say 140 million subscribers. Directory-based partitioning, a loosely coupled approach to work around the issues mentioned in the above scheme is to create a lookup service which knows your current partitioning scheme and abstracts it away from the database access code. To find out where a particular data entity resides, we query the directory server that holds the mapping between each tuple key to its database server. This loosely coupled approach means we can perform tasks like adding servers to the database pool or changing our partitioning scheme without having to impact on the application. Now let's focus a little moment in the partitioning methods to the partitioning criteria. Let's say we use key or hash-based partitioning. We apply a hash function to some key based on the entity we are storing that yields the partitioner number. We saw that in the DynamoDB data, how we apply the hash function in order to determine which raw the data will go. Let's say we have 100 database servers and our ID is a numeric value that gets incremented by one each time a new record is inserted. The hash function could be, let's do the ID module of 100, which will give us the server number where we can store in that record. This approach should ensure a uniform allocation of data among serves. The fundamental problem with this approach is that effectively fixes that total number of database servers since adding new service means changing the hash function, would require redistribution of data and downtime for the service. A workaround this problem, of course, is consistent hashing. The other way is list partitioning. Each partition is assigned a list of values, so whenever we want to insert a new record, we will see which partition contains our key and then store it there, for example, can decide all users leaving Iceland, or Norway or Sweden or Finland, Denmark. They will be stored in what we call the Nordic countries table or partition. Finally, Round-robin partitioning algorithm is a very simple strategy that ensures uniform data distributions with N partitions and the I tuples is assigned to a partition. We do like an I module N, this is the very simplest round-robin partitioning algorithm. The composite partitioning is we combine any of the above partitioning schemes to devise a new

scheme, for example, first applying a list partitioning scheme and then a hash-based partitioning. Consistent hashing could be considered a composite of hash and list partitioning where the hash reduces the keyspace to a size that can be listed.

>> So let us discuss now how we can scale data systems on the Cloud. Back in the days, back in early 2000s, read scaled out by adding replication slaves to a master database, so this is mainly for relation database systems. So what you would do is you'd have a user that will write some data to the master node and then the data would actually be replicated to different types of slaves, so master node and slaves would have actually the same types of data. In this analogy if the slave went down, for example it was terminated, then the master will actually still have the data fine as the writes would go there. Then the reader, which is usually some user, will actually, instead of reading from slave 0 will now read data from slave 1 or slave 2 and so forth, so it would feed off on other nodes. So this would work well in the early days of the Internet because there were a few people that were publishing data to the Internet, a few experts, and then a lot of people that were actually reading data. So effectively writes with the master nodes and then reads with the template slave node, and then a slave could actually also have some transaction. So you will commit some data and unless data is being replicated then you acknowledge the transaction and so forth. Now the problem becomes interesting because now if your slave nodes are across different regions, your transactions may have very long latency. At the same time, you may actually have slaves in other regions just to have backups. So at small-scale things are okay, with relational database systems, having a single node to write data, the node can become overloaded with high write throughput. So on a small-scale it's fine, but once you start discussing about high write throughput cases, then having a single node where you can write the data, it becomes a bottleneck. At the same time nodes can be terminated or restarted by the Cloud provider, you don't know when this happens. RDBMS are what we call CP systems (consistency and partition tolerance systems). They are not designed to be highly available, hence when the master node gets terminated, we need to the re-elect a new master node out of the slaves, which means that now you're blocking all the writes. You are not allowing any writes to happen because you need to wait to re-elect a new master node, otherwise you may end up in a system that is not consistent. Within the last 10 years we've seen also the growth of online platforms, Facebook, Instagram, Twitter, and so forth. A lot of machine learning algorithms that are being run by the data that you are actually publishing online. So that means now we are not being in a case where we have a few writes and a lot of reads, but we're going towards patents where you actually can have a lot of writes. So let's now think two types of designs, an optimistic design and pessimistic design. Now pessimistic design, design with high concurrency, you effectively punish 99.9 percent of the users all the time. But you can achieve a high consistency, but then the problem as I said is you can achieve a higher latency, that results in diminishing the user experience. In an optimistic design you can trust your data store, like know your business and your application, always ask yourself, "Is it really that important to have consistency?" Then figure out maybe a backup plan for that. So one of the database that have actually been very famous for the high availability and actually at scale is Cassandra. It is one of the implementation of the Dynamo protocol in Cassandra. Cassandra is not now strictly consistent, it is eventually consistent, so the data will eventually get there once you publish the data. But what does eventually mean? Eventually means that they could get there a day from now, a minute or a second from now. Actually, what happens in many cases the data get in the other node in milliseconds. So Cassandra, you can think about that as a multi-master system, where you can have like multiple people writing nodes differently, to different elements of the database.

>> Let's do a little deeper dive on the DynamoDB. We discussed about eventual consistency and how eventual consistency is being used in different areas from financial decisions to Amazon to Netflix, to really many and large companies even nowadays. Dynamo is the name given to a set of techniques that when taken together can form a highly available key-value structured storage system or else a distributed data store. It has properties of both a database and a distributed hash table. Let's get into more detail about that. Assuming that you want to represent something, like you want to buy walkie-talkie from Amazon. Amazon has a specific ID, an identifier like a number, that is associated with the specific walkie-talkie. So when you go and pay for a walkie-talkie, the E-system gets back with an ID and says, "Here's the ID of the walkie-talkie." Then you say, "Oh, give me the walkie-talkie." Then what you do is you do what we say, I get operation of the ID so that you can get the walkie-talkie. This is how effectively a hash table works and this how a key-value store works. So let's see the Amazon challenges. It's the largest e-commerce site in the world. It serves millions of customers, I mean close to billions, at peak usage time with using thousands servers in many data centers around the globe. Especially during peak operations, Amazon procures huge infrastructure to be able to help with the demand. Strict operations requirements on Amazon's platform in terms of performance, reliability, and efficiency. So a slight outage can have a significant impact on the financial consequences and impact to the customer trust. The first thing that you want to do is you want to make sure that your system is not down. If Amazon goes down, then people are going to start filling out, they will go on Twitter and start writing about that. They'll do the same when they go to Netflix or Facebook or any of these big platforms. Because as more and more the world becomes connected, we're very much dependent on them. Which means that whatever we design as a system has to be scalable. It has to support continuous growth of customers as they grow from a few thousands to millions of customers. This holds for effectively any consumer-oriented platform right now. Amazon uses a highly decentralized, loosely coupled, service oriented architecture consisting of hundreds to thousands of services, and all these services what they need, is they need to storage. So customers should be able to view and add items to their shopping cart even when disks are failing, or network routes are flapping, or even when data centers are destroyed because of tornadoes or even any other natural disaster. Then the service that is responsible for managing shopping carts requires that it can always write and read from that data store. So you can write means that you can add items to the shopping cart while reading means that you can read the shopping cart so you can check it out and pay for that. What is Dynamo? Dynamo was initially a highly available distributed key-value storage. So it has two interfaces, a put and a get. Put means that I will write some data. Get means I want to get some data. What it does, it sacrifices consistency for availability. If we go back to the cart filling, what we discussed was that it is an incredible and partition-tolerance system like an AP system. It provides storage for some of the Amazon's key products, like the shopping carts, bestseller lists, and so forth, and it uses a series of well-known techniques to achieve scalability and availability. Some of these techniques are, for example, consistent hashing, object versioning, conflict resolution. But what we are going to do, you can actually go and read the paper for Dynamo to get into the details about that. It is a set of techniques to form a distributed key-value structure system. After the Dynamo protocol was published in 2007, there were a few systems that were developed based on Dynamo. DynamoDB was one of them, Cassandra was another one, Facebook and so forth. There's maybe other database that where it came from that. The scale. Amazon is busy during holidays. Tens of millions of requests for three million checkouts in a single day. Session state about $100,000 of concurrent active sessions. So failure is not very common. Yet a small but significant amount of servers and network failures happen at all times. So customers should be able to view and add items to their shopping cart, as I said, even when we have these important disasters at even high-scale. The flexibility. We need to have minimal need for manual administration. So you cannot call somebody to fix an issue when I was almost out. It has to auto heal itself, its subsets. So nodes can be added or removed within the data system without manual partition or redistribution. So what happens when a node goes down, the data system could be able to sustain that. Applications can control availability, consistency, and cost-effectiveness. So can developers know this upfront? Can it be changed over time? The question is now, how do you propagate this back to the users effectively? So assumptions. Assumptions that the data which has a very small, very simple query model or I put and I get. The values, first of all, are about one megabyte of binary objects. Then no ACID properties. So we have weaker consistency, no isolation guarantees and single key updates. Efficiency. It has stringent latency requirements up to 9.9 percentile. So percentile vacancies are fixed personalities, half of the users like that. An IT person is like 90 percent of the users have been doing that, and 99.9 percentile means that 99.9 of the users are bigger especially sample. If we will say 99.9 percentile latencies are 50 milliseconds, that means that 99.9 percent of the users or requests are actually getting 50 milliseconds of latencies. Of course, that the deployment itself is done in a number of data centers. It's not a hostile environment where you don't have control. Security can be a little more relaxed or handled by another system itself. As we said, a Dynamo is an eventual consistent system. It has to be always writable. So whatever happens when somebody

adds an object in the shopping cart, the object must be there. What it does, it pushes conflict resolution to the reads. So if two people add the same object, or for example, where two people are trying to check out the last object from Amazon, let's say the last cucumber. So what happens is both will add it, but at a later point of time, when they try to read it, it will say, "Oh, I'm sorry, somebody else had the cucumber, you cannot now check it out." It's application-driven conflict resolution. It merges conflict with some shopping cart or Dynamo enforces what we call last-writer-wins. So effectively, when two people try to buy a cucumber at the millisecond level, Amazon will try to say, "Oh, you know something? The customer Number 1 was the one that actually put it first." That's why it says last-writer-wins. So the question, does it work? Of course, it does work on a high scale like apples the user.

>> So let us discuss the service level agreements. Service level agreements is something that we generally use in a lot of services on the cloud. I'm going to try to explain them a little more in detail in this subsection, with a hope that these parentheses from the Dynamo protocol will get us understanding about some of the techniques have been built within Dynamo to make sure that we satisfy some of these service level agreements. Service level agreement is a document where the cloud SLA is effectively a document. It's an agreement between a cloud service provider and a customer that ensures a minimum level of service is maintained. That means that if a service is being offered, you want some guarantees that the service will be offered based on how it is described. In order to do that the company and the customer signs a service level agreement. It guarantees the levels of reliability, availability, and responsiveness to systems and applications, while also specifying who will govern when there is a service interruption. this is more like a protocol, a service goes down, what happens? This needs to be described within the same set of agreement. Like the costumer will observe it, it will not find the cloud provider and the cargo will actually act on that. So why do we need an SLA? A cloud infrastructure can span geographies, networks, and systems that are both physical and virtual. While the exact metrics of a cloud SLA can vary by service provider, the areas covered are uniform, for example, a volume and quality of work, including precision and accuracy, speed, responsiveness, and efficiency. So what this means, it means that at a high level, the SLAs are fairly similar across the board. There will be exceptions always. For example, how long will the system be down? So if the SLA says 99.9 percent of the time the system is up, that's an SLA, that this provider provided. The document aims to establish a mutual understanding of the services, prioritized areas, responsibilities, guarantees, and warranties provided by service provider, and warranties is very important here. Because if the provider does not meet the SLA, then there has to be some impact. So how does the provider captures those SLAs? It has metrics, and responsibilities among the parties involved in cloud configurations are very clearly outlined, such as the specific amount of response time for reporting or addressing the system failures. There are many metrics you can do that, and there is a mean time to failure, and that's one metric, there is the how long the system will be down, and so forth. If the provider now violates, there are financial penalties the providers must pay for failing to live up to the guaranteed terms are also included. For example, how much you're going to pay back if the service is down? These penalties are often in the form of credits for service time. Now, think about it. Nowadays, there are companies which has multi-billion and hundred of billion valuation like Netflix, that is based, the control play is based on AWS. So if AWS goes down, that's a significant business impact to Netflix, which means that somehow it has to guarantee that AWS will actually provide the proper SLAs so that it can run the business, and this works for every customer. So SLAs are very widely used even across web services, even those that don't use cloud, like Facebook has it's own private data service. Sub-services must meet very strict SLAs, an example would be 300 millisecond response time for 99.9 percent of the requests at peak load of 500 requests per seconds. So effectively what this is, is you have to satisfy an SLA at a specific percentile, and this is what we discussed through the example. An average case SLA are not good enough, and the reason they are not good enough is because you may have spikes, like a spike of let's say, one or two requests that are causing substantial business issue. Because now the costumers are cannot access the service. So if you have let say one out of the 100 requests that are failing, the sign-ups, when somebody needs to sign up on a service, that means you are losing 10 percent of your customers. At the same time, it's not that you have SLAs for the whole thing, you have SLAs for each of the substances, and then sometimes I don't think a single service can actually make request to 150 services, which is a lot. So if each of them is failing one percent of the time, now, the probability that you work the whole service eventually is very high. So application can deliver its functionality in a bounded time. Every dependence in the platform needs to deliver it's functionality with even tighter bounds. We discussed that about the critical path, this is now how the latency within the critical path is important, becomes like a part of a contract. Design considerations. So when we design a system like Dynamo, we have effectively to sacrifice strong consistency for availability. Other [inaudible] SLAs will be hampered. Conflict resolution can be execute, or the read instead of the write, for example, every write goes there. What I mean by that? I mean that if the write goes in, then that means the surface is up, that means that you can actually satisfy the SLA. But if you design your system through like a relational system, the problem then becomes that some of the writes may fail. Other figures of Dynamo includes incremental scalability, where you can add notes based on demand, and this is important for these SLAs as well. Because if the demand increases, we need to be able to increase your ability to serve the increased requests. It provides principles like symmetry, you can allocate requests across all different replicas, decentralization, and heterogeneity.

>> Now, let us go back to the Dynamo Partition Algorithm. So we are going to go a little deeper on the algorithm itself. I hope that many of you have read the reading assignment, which describes parts of the Dynamo protocol. So you may be already aware of these elements. If you haven't, please go ahead and study that paper even after this lecture. One of the things on a system like Dynamo, it uses something we call consistent hashing. What happens is, say you have some input like a text or some information, and then what Dynamo does, it applies a hash function. So a hash function is a form of randomizing the input in some sense. Hash functions are very commonly used in cryptography. When you say, I want to create this, it takes a little bit and then it produces some random outputs. This is simpler methodology. It uses, again, a hash function. The only difference is, within the encryption domain, we use a hash function that is hard to actually reverse them. They're very advanced, computation intensive, whereas in situations like databases, we don't need really too advance hashing functions that take a lot of computation. So what happens is that assuming that you have nodes like A, B, C, D, E, F, G, and then you have a key like K that you want to store it. So what happens now is, you have a range from, let's say 0-100, you take an input A or take an input K, and you hash it. So the output will be a number between 0-100. Assume that number is, let's say five or 10 and assume that the range between A, B is from 0-10. So that means that the key K will actually belong somewhere in between A and B. Therefore, the data will be stored probably, let's say in A, in that node. So that's how, at a high level, it works. So virtual node, it's not categorically possible for more than one virtual node. Let's discuss about virtual nodes. If a node becomes unavailable, the load handled by this node is evenly dispersed across the remaining available nodes. So the notion of virtual node means that now, we don't have a single range that a node handles, but rather it becomes virtual node. So it handles multiple different ranges. So nodes B, C, and D store keys in range A and B, including K. So that means that now, these three nodes include everything within A and B, so that becomes a larger virtual node. When a node becomes available again, the newly available node accepts a roughly equivalent amount of load from each of the other available nodes. So the number of virtual nodes that a node is responsible can be decided based on its capacity, account for heterogeneity in the physical infrastructure. So what this means, it means that assuming you're deploying Dynamo in an infrastructure, what do you have? One, it's something that has two eyes, the hard disk, then the other ones. You want to make sure that the corresponding capacity of a virtual node is evenly distributed based of the capacity of the nodes in infrastructure. So let's see variant of consistent hashing. Each node now is assigned multiple points in the ring, like B, C, D stored key range A, B. The number of points can be assigned based on node's capacity, as we said. If node becomes unavailable, load is now distributed to other nodes as we discussed before. So assuming now you store data key K, the coordinator for that may be, let's say node B. Node B maintains a preference list for each data item, specifically nodes storing that item. So what happens is preference list skips virtual nodes in favor of physical nodes. So D stores A and B, B and C, C and D. So effectively, what happens now is that the data from B is now replicated to C and it's replicated to D. So D now stores everything from A and B, B and C, and C and D. That's the whole range it handles. E now handles everything from B to E. F handles everything from C to F. G handles everything from C to G, and so forth. So then what happens if you receive the data? Then one node becomes a coordinator and make sure that it replicates the data to the other nodes. Each data is replicated at N hosts, in this case, at three, it has this notion of preference list. So that's the list of nodes that is responsible for storing a particular key, as we discussed. Data versioning. So DynamoDB also uses data versioning. So let's see about data versioning. A put that is a right call may return to its caller before the update has been applied at all the replicas. We do that because of low latencies, otherwise, it will be like a transactional system. A get call may return many versions of the same object. That is because when you do a put, let's say you add an item to your shopping cart, that ends up becoming a write in the database. Once you try to fetch it, maybe the data have not been replicated, or maybe there has been a network partition that didn't allow the data to be replicated. So the challenge, an object having distinct versions sub-histories which the system will need now to reconcile in the future. What means is you may have, let's say you give it a number from one to two, now the aggregate of the master node equal to 2, but all the other nodes are still at 1. So what Dynamo uses, it uses the notion of vector clocks. A vector clock in order to capture causality between different versions of the same object. So in a shopping cart example, Add to cart and delete an item from cart operations are translated as put requests in Dynamo. Add to cart operation can never be forgotten or rejected. When a customer adds items to a shopping cart and the latest version is available, the item is added, the older version and the divergent versions are reconciled later on. Dynamo treats the results of each modification as new and immutable versions of the data. Immutable means they cannot be changed. Most of the times, the new versions subsume the previous versions. Version branching may happen. In the presence of failures combined with concurrent updates, resulting in conflicting versions of our object. So what may happened is, now you to a put in a node and another node that's a write or an update, now you have federally conflicted nodes, you cannot really reconcile. In this case, the system cannot reconcile multiple versions of the same object and the client must perform what we call a reconciliation.

Example is merging different versions of a customer's shopping cart. Add to cart operation is never lost. Deleted item can resurface. Let's have a single example in the case. Assume you add some data in your shopping cart from your laptop. At the same time, you sell your cart with somebody else. There's some data in the shopping cart from their smartphone. That results now in version, different updates happening from two different systems effectively. So the way we do that is through vector clocks. A vector clock is a list of node counter pairs. Every version of every object associated with one vector clock. If the counters of the first object clock are less than or equal to all the nodes is [NOISE] in the second clock, [NOISE] then the first is an ancestor of the second and can be forgotten. Otherwise, what happens is that the client has to do the reconciliation. Let's go a little deeper around the algorithm of the vector clock. So assuming now a client writes a new object D1, the node Sx that handles the write for this increases, the sequence number. Until the client updates the object D1 and D2, D2 overrides D1. There might be replicas of D1 lingering at nodes that have not seen D2. So in this case, for example, you may have a write handled by Sx, you may have from one, it became to two. [NOISE] Then you have a write. Now, for example, you have another increment that is now handled by Sy and another that is handled by Sz. So what happens now is the write, since it's handled by two different nodes, you may have diverse information in the system. Effectively, if the next write now is reconciled Sx, you now may have three different versions in the system. So you need to find ways that will need to reconcile that because you cannot establish a relationship. In this case, with D1 and D2, within that same relation because it was handled by the same node, that is Sx. But once the data handled by different nodes, it's hard to handle some source of causality so you can figure out what is the most recent update. Eventually, what happens is you need to expose the divergence of the put operations up to the customer or make a random choice. A random choice obviously is the wrong way to go, because if you make a random choice, you will deal with a lot of problems. But what happens most of the case, the system, the application may see three different versions. So you can route its request through a generic load balancer that will select a node based on load information. So you can use a partition-aware client library that routes requests directly to the appropriate coordinator nodes. So this is the most important. Assuming now you write some data to a node Sx, and all of a sudden, what happens is you have a generic load balancer or a client library that says Sx is overloaded. When I push y, all of a sudden, now you cannot establish causality. But you have to do that because sometimes, essentially, it become too overloaded, so you may have to distribute the workload. If you do that, now you have to have the promise of eventual consistency. One of the reasons you can solve this problem is, as we said, with vector clocks. There are many other ways you can solve the problem, but if the problem cannot be solved, you have to propagate up to the customer as we said before. Sloppy quorum; reads and writes is the minimum number of nodes that must participate in a successful read and write operation. Setting R plus W greater than N yields what we call a quorum-like system. So what happens is [inaudible] now a divergence, assuming that you write the data in three nodes as we said in the beginning. So you write the data in three nodes. All of a sudden, two of the nodes have wrong information, or let's say two of the nodes have correct information, one does not. So if you try to say, I'm going to do a quorum write, that means that as long as the write has been accepted by at least two nodes out of the three, then the write will be accepted by the system. If it doesn't, then the write will be accepted by the system. In this model, the latency of a get is dictated by the slowest apps. So if you have, let's say, three columns of data, and you won't acknowledge two nodes, now over [inaudible] data will be higher because you have to get acknowledgment from two nodes. For this reason, R/W are usually configured to be less than N to provide better latency. So assume that N is equal to 3. When A is temporarily down or unreachable during a write, send replica to D. D is hinted that the replica is belong to A and it will deliver to A where A is recovered. Again, always writeable. What it means is that you can also hold off some hint, that the data have been deleted and you can update the data back to A. What may happen though is that there might be another update date before D sends the data to A. In this case, you have to use some, let's say last writer. Let's say you have some timestamps, that will actually win. So if there is an update date before the sends data back to A, then that is what you effectively have, do not update the date. So other techniques that DynamoDB uses is replica synchronization like Merkle hash trees or membership like Gossip. I'm not going into detail about that. I think the whitepaper which I posted light should have a lot of these details. Dynamo was developed in Java. Java is not always the best technique for web services, though it has a lot of easiness because of JVN issues. Locally persistence component allow for different storage engines to be plugged in. So you can use Berkeley database, MySQL, BDB, and so forth. So you can use different storage engines [inaudible]. Persistent store either Berkeley DB transactional data store, BDB, MySQL, or in-memory buffer. All are in Java. The common setting for N, R, and W for Dynamo is 3, 2, 2. Results are from several hundreds nodes configured as 3, 2, 2 nodes [inaudible] on a single data center. Conclusion, Dynamo is scalable. Easy to shovel in more capacity at Christmas. It's simple, you have two APIs, get, put. It's flexible. You can set N, R, W. It's inflexible. Apps have to set to match the needs. Apps have to do their own conflict resolution. They claim it's easy to set these. This mean that they have many intersecting points. So bottom line is it constricts, builds, and flexibilities depending on how you view the system. So

sometimes application have to set it, it becomes complicated, so they don't like that. But it has to be a collaboration between the app developer and the DynamoDB engineer. [NOISE]

>> Welcome to the new module. Let us discuss the battle between SQL versus NoSQL. This is relational versus non-relational systems. One of the main things we need in any cloud computing environment is the ability to interact with some form of data storage, whether this is for airline reservations, bank accounts, scheduled pizza deliveries, and so forth. That means that as most of these platforms tend to become web right now, we need to support them by the ability to scale on the cloud as well. One of the primary design goals that we did in order to move to the cloud was because of the scalability. If we didn't care about scaling, we wouldn't have gone to the cloud. One of the reasons that we move to the cloud is because the cloud provides horizontal scalability, that means you can add more instances to scale your services. But even if you don't need to add more instances, you can improve your instance type so you can effectively vertically scale. Some of the things that in a physical infrastructure will actually require a total update of the infrastructure. Our cloud applications data harmony has to scale as well, which means that if we scale the computational part of our applications that is the services and microservices that the customer state to interact with but we don't scale the data, handling the aspect of that, that obviously we will get a bottleneck, and then that may not be a position we want to be. A very simple example here is if you scale your services that add information to the shop and cart in any of these shopping websites but they say, dive your database, that actually stores the data at the shopping cart does not scale, then the problem is that you cannot anymore scale your services. At the realm of these, there are effectively two storage kingdoms in the data systems. One is the non-relational systems and the other is the relational systems. NoSQL or SQL, you probably going to hear both terms interchangeably across these presentations. The common data structure in non-relational systems is blobs, tables and queues in your later phase, we are going to discuss these in detail. There is no consistent interface so which means that most of the vendors, they deploy their own interface on how one really access the database. They tend to scale very well, and the reason because of that it's actually they give up some of what we call the data integrity aspect. Most of these systems don't provide transactional guarantees, so whatever integrity the program requires is the responsibility of the program developer to deal with. If they don't need to get a transaction that the user has to deal with something with these capabilities. But this is what we need to give up and we discussed a few of these things in the CAP theorem was one. In the relational kingdom, there are classics table database like Oracle, Microsoft SQL, MariaDB, Postgres, and so forth. They provide data integrity services such as not allowing the customer to be deleted from a customer table as long as an order that order table references that customer. This is effectively the transaction. Once you have something that is pending you should not be able to delete something else. The support complex queries involving joins across tables, and this is one of the advantages of relational systems. Now you can actually have very complicate queries and effectively SQL became from a very simple language to a very complicated query language, because people will do very complicated queries to serve the business needs. It's harder to scale especially at multiple machines, and the main reason is because if you start scaling horizontally, you will start losing your single point of handling some of your transactions. Often they contain legacy data for applications that need to move to the cloud. At the real beef, we start comparing SQL versus NoSQL. We see that the non-relational database does not incorporate the table modeling. State data can be stored in a single file, in a blob, in a queue, or whatever it is. But in SQL space, you have a more stable structure and you can create relationships between those tables. For example, you can say, this is the product table, this is a prize table, and this is the ingredients so you can actually relate the information between themselves. Let's now discuss a very simple example of an SQL example. Assuming that you are creating a blog, the table for all your posts, each post in that table has a row, it all has something like a numeric idea that text of that post. This is an example of where you actually create a blog and its posts in that table is a row, and then you have an ID for each of the rows of that post. When you want to read the blog post, the URL in that address bar of your say something like give me this post number 1. You do a query says select everything from the posts with an ID equal to 1 and then you get back the first box. Once you get back effectively the software itself, the web browser renders HTML, and then it presents back that post. The relational model typically comes into play when you visit a blog that has some comments. That comments table will most likely have the same ID column and a column called body or something like that for storing text of the column. In a common thing is now you could create another table where that column table is now correlated with the post table, and then you can say, I won't let say post number 1, the column number 1. You say, select this comment from this table and then give it up sorry disposal off this table and give in the comment from the other table that you have established a relation between comments and posts. This table will also have a column called something like post ID and what gets stuffed in the column is the ID of the blog post that this column relates to. Effectively you create another, as we said, table, that within the information of the comment you have also the post ID, which backreferences the post table. Now when your reader comes by the blog software turns around to the database and asks for two things. Select everything from the post table 1 where the idea of the post is equal to 1, and select everything from the comments table where the post ID is equal to 1. Now the second query is if you are lucky enough to

write something that people respond to, will return a list of comments, an array if you will, that your blog software will then convert to HTML and append to your blog post in the form of the comments section. So which means that you do a single query, HDB, your HTML that you respond back, is if it includes the blog post at the comments section associated with that. This is a great example for a blog and many blogs have actually designed like that because you don't have billions or trillions of comments or posts that have in the blogs. In the NoSQL, example is the request comes in blogging software turns around to database and says, please give me back the specific post, anything related to it. In this case, listing of comments, since we're not forced to be too uptight about having to define how the data is structured. What if we want to tag that post with an arbitrary number of categories? No problem, let's stick them on the same document and when block software says, give me everything you'll post to add, the tags, the comments, and any other randomly associated data come back with it. Effectively what we do is we can do a scan on the database says give me everything that relates to this specific worst. If you see in this example, obviously the table format fits a little better than let's say the blog format. This is one of the things that you need to identify how your data model is going to happen. In a grocery case, you can model the grocery list in a lot the same way, like a piece of paper you write the items you want on there. But let's say for the purpose of this example, you figure out after a couple of months that you want to keep track of how many loaves of bread you actually brought last year. Well, probably with the non-rational model you might literally have to go through every list and count each loaf individually whereas had you modeled this in a rational way, you could get that count back almost instantly. This is another inefficiency of NoSQL. If we tried to go back to SQL vs NoSQL, the pros is in relational data model is that it is easy to use and set up. It's universal. Everybody uses SQL model. Already started stated goal of stages or the query it may be tailored to the needs database. But fundamentally, the user doesn't really care about all these complexities. It's good at high-performance workloads, especially when you have high-performance meaning up to 10,000 or 20,000 writes per second but above that doesn't scale very well. It's good at structured data like tables and so forth. The SQL side of that, the glide-like collide with this type of technical shifted aside that decide the structure of the database. It is going to be difficult to scale. The process of the checklist there is no investment to date design model. You can wrap it developed over cycles. Generally the APIs, but even a simpler than SQL editor runs very well with the cloud because you cannot constantly take the schemas and you have no problem about that. But it [inaudible] says treated for interconnected data, what you need is effectively relations between tables, for example, say the blog and the comments. The technology still virtually there's a lot of systems out there but still virtually, it's going to have a slow response time in some cases but can scale with a lot of nodes when we add a lot of wiggling scale horizontally with a lot of nodes.

>> [NOISE] The first steps we learned about the non-relational and relational model. Today, in this module, we're going to go a little deeper into the relational model. When we went back and discuss the CAP theorem a few sections, we discussed that SQL databases emphasize in the ACID, that is in the atomicity, consistency, isolation, and durability. You know what a slow SQL systems focuses on the BASE, basically available, soft-state, and eventual consistency. These are both based on the Brewer's CAP theorem, which is visualized, as you can see on the right side, a common definition of CAP, as we said this in the face of network partitions, you can't always have perfect consistency and availability. You need to effectively choose two out of the three. As you can see this example, like systems like Mic SQL, SQL Server, MariaDB, which are relational systems, are focusing on consistency and availability and consistency, where systems like Cassandra, Riak, CouchD, they focus on AP. They are not like relational systems. When you see consistency, you can think about some form of a relational system. These are associated familiar database like Oracle, Microsoft, SQL Server, Postgres. They made up of tables containing typed columns, schemas at raw data. Used when relationships between tables are necessary and complex. For example, an order contains a reference to a product and a customer. What happens is that you have these multiple tables and you tried to define connections between the tables that store the data. The customer contains a reference to an address table and a payment source. Database system itself enforces data integrity, for example, you can't delete a customer when an order refers to it. That means that you have some transactional semantics that happened in the background. Relational storage systems have historically been the bottleneck of large systems because as you try to horizontally scale, you can horizontally scale that no one handles the transactions. You need to scale up from a single node, often a large fast one, but have one more trouble scaling out because of need for consistencies which means that, you can effectively have more and more [inaudible] ACID. That also, you have to keep in mind that the more [inaudible] the more data you store in this master node, the lower your availability if this node goes down because I have a substantial amount of data going down before you are able to reelect and you master to huddle your rights. Having said that, most legacy data is this type of storage, so you won't lose them at any time soon. Relational storage are not gradual way. They serve a substantial business reason and they will continue doing that. Let's see here an example where you have a car, you have the CarKey, the MakeKey, the ModelKey and then you have the color, the Make table and the Make. You have these four tables. As you can see now, you can establish relationships between the color of the car and the car itself or the make model or the make, right? Therefore, if we look at the query, say, you'll find me all the cars produced in 2003 with color red and tire and make Nissan. In a relational database, a table is an organized set of data elements. That is the values. Using a model of vertical columns, which are identified by their name, and horizontal rows, the cell being the unit where a row and a column intersects. As you can see here, you say the the student, John Smith, with an ID 84. Now, the ID says they do a tennis and they have paid $36, they do swimming, and they have paid $17. Now, you can establish the relationships between the systems. An SQL Joins clause combines records from two or more tables in a database. It creates a set that can be saved as a table or used as it is it, all right? A JOIN is a means for combining fields from two tables by using values common to each other. In this case, for example, we have recipes, which is the table. We have the ID which is an integer, the name which is a character for 100 bytes. Then we have category ID, which effectively points to the category table, which also has a character 255, then another key that effectively points to this table. You can see now you have multiple tables that are appointed from the same entry, which is the recipes.

>> In this module, we'll cover a little about the non-relational models or else the NoSQL [NOISE]. NoSQL databases is any of the modern databases that essentially give up the ability to do joins in order to be able to avoid a huge monolith of tables and scale. Some of the examples are key-value stores like the Vanilla Dynamo paper in 2007 or even Redis which is a key-value store with a very simple API in advanced data structures. Some others include HBase and Cassandra. While you may think about that Cassandra was based on the Dynamo paper in terms of high dozen application, it does use coding family underneath it. Another flavor of a NoSQL database, is document-based. An example is MongoDB, which is one of the most actually famous NoSQL databases in the market. The NoSQL databases usually has some form of a flexible schema, that means there are no rigid tables and no rigid N by M structures that can be caused by joins across tables. Flexible schema means that at any point of time you can increase any of the fields or add new fields within the schema that you have. An example would be when you are trying to have let's say Amazon, represent a product and all of a sudden one day they decide to add some new field for example, the metadata of a specific product they can do that very flexibly within the NoSQL databases. Why NoSQL in the last five, six, maybe eight years has become a big market shift from the relational system. The problem was that within the Cloud era we see that the datasets started to become too big, having a single node that will accept all the data was not good enough. There are different ways that you can achieve scalability. [NOISE]. The reason this happened is because now you had hundreds of thousands of visitors in a short span and that is a massive increase in traffic within the database itself. Effectively, there are many part business started through that. One of them was microservices and the others is splitting the NoSQL data systems. Now developers begin to front RDBMS with read-only cache to offload a considerable amount of read traffic. This was the first level of redirection, but once you actually add some read only cache, let's memcache or Redis, the problem is you cannot anymore do transactions on the cache itself. Memcache or even integrated caching mechanism within the applications was another way you can achieve that. For example, you can have a huge house index within your applications, like a read memory index or distributed replicate that close objects over multiple nodes. But as data grow, the simple memcache/MySQL model for lower costs startup start to become fairly problematic because now you have lots of data, lots of traffic, and a single master-slave system was not good enough. The main objective of NoSQL is that ingestion of data that has unknown or undetermined structure. This is what we discussed before as the schema structure. It is relatively easy of scaling with increase volume and velocity of data because now you can add more nodes and horizontally scale your data systems. It has the ability to analyze large volumes of data significantly quicker than traditional SQL databases, and the reason is because once you have joins then you start doing scans across joins it becomes very problematic because of the nest things that you have within a database. In the NoSQL side, you don't really have nest thing. You have [inaudible] things, but you don't have joins, which makes it complicated. It again finally it provides the capabilities to derive value from data, whether it is big or large data. NoSQL is actually Not Only SQL. It provides Insert only no UPDATE/DELETE. There's no joins, thereby reducing query time. This involves de-normalizing the data. It does not support SQL. It supports the old like very bad vental driven Clyde protocols. It has no adherence to ACID properties like an atomicity consistency, isolation, durability, which actually makes a few things simpler but a few things a little more complicated. So SQL, not only SQL, is schema free, embraces de-normalization of data. It provides greater scaling capabilities. It has a very simple design. You don't really need to have advanced transaction, multithreaded support. No need to have advanced query ages. Storing objects that represent your domain. It gets great for unstructured or semi-structured data. So actually, when you don't need tables and rows, in unstructured form, that's where those SQL becomes very, very interesting as a product. The initial seeds of the NoSQL era started by Bigtable, which was one of the papers that Google wrote, the Dynamo paper that was published, which is a distributed key-value store. But also it was important that they included the gossip protocol which is subsets embedded with many other data systems like cassandra. The CAP theorem as was formalized by Eric Brewer and effectively it became loud between BASE and ACID, these two of these properties based on what he did. He did a base, NoSQL system or you did acid guarantees which you get from relational systems. One of the main issues of relational systems is of course that you can scale the RAM, you can scale the CPU, you can scale the hard drive but at the same time, this is called vertical scaling. At some point you actually hit a limit. On the Cloud what you really need to do most of the time is use your horizontal scale. You can add more hardware or more instances or more virtual machines, or even more containers in some cases to scale up your data systems and this is what we call horizontal scalability. With relational systems, you can only get with a vertical scaling. With low relations you can actually use horizontal scale and vertical scale as well. As we said, there are different flavors within the noSQL side. We move from the tables and rows. That also the corners like a tabular format to the analytical OLAP towards a key value called the family graph and document flavors. Of course, there are different types of databases that you could use. Here there's the column stores which have read and write extensions. Some of these are HBase and Cassandra. You have the

document stores, which you have a significant blob of data that you would store, like sensation file, an example for those is Couchbase or MongoDB. Then you have graph databases where you have some complicated high level of nesting, which is like spatial data storage. Neo4j is actually the one which is the most popular one. Then you have auto key values those like Riak, Redis. Riak does not exist anymore which actually provides you to some of the high, very fast and very stable access mechanisms. [NOISE]. As we start to moving from relational to non relational SQL we see the traditional format schema of the data, where you have like a post id and the title. You don't have much flexibility. [inaudible] have a lot of flexibility. So in this case, you can have both post a title and a different types of comments. But then you can have also another post where instead of having the comments, the title too you actually have the images. This is what we call by flexible schema. You can save the schema based on what the application needs. Within a database, you can have, for example, the something that records the comments of a blog post, as well as the id's of the images that you can face from an objects thoughts. But whatever you move it comes with a few disadvantages. The fundamental disadvantage of NoSQL is lack of standardization, support, and maturity. Maturity is not that much of a problem anymore because some of these databases have increased in terms of maturity. Non relational database do not use a Structured Query Language, the SQL. There's no unified standard language it's either it uses it's own way to actually access the data. This makes it difficult to migrate the process over to a NoSQL. Actually, you don't only need to migrate the data, but you also need to modify the application. If you still use SQL database, as in you have written your application using SQL, which is just a common query for math of course, most of the systems whether to use MySQL or Postgres or any of these systems, the application does not really need to change substantially based on the data system. It has to change a little but not substantially. However, once you move to NoSQL systems like Cassandra has its own query language, they call its CQL, Cassandra Query Language, but I discuss its own language, Mencast has its own key-value APIs. Every one of them have their own APIs and clients. Some of the NoSQL database requires specialized expertise. For example, once you start scaling to like horizontal scaling to hundreds of nodes Cassandra it becomes very difficult. Whereas in actual database, you can have a DBA where the fundamental aspects remain the same across most systems. Then for many companies may fall in the trap of using Open Source S NoSQL. Because using an open-source on an NoSQL it's fast we can, and what you end up, when you become at scale, the problem becomes a little more complicated. Let's now see a little bit differences. Here we have on the left side that snippet of differences between SQL and NoSQL. Actually you can see the operations get a little more complex in the JSON format. Here, you can insert to a book these values whereas here you can do the same thing with adjacent document, which is okay, simply good. When you try to do an update, it's clear, it's an updates whereas here you have to do an update, you have to give the ISBN, you have to give the set, you have to give all of these properties in a select query, select title for a book where it becomes more natural to express that, compared to like expressing in this form of a JSON format. But there some advantages of NoSQL like flexibility which is easier to manage and more adapt at dealing with newer data models. Then for NoSQL can fit very specific needs of the company and it can be really faster to scale it up. It's highly scalable especially at low cost. Many NoSQL options are open source making them more affordable option for smaller organizations. You notice they speak in some of SQLs have the same [inaudible] like [inaudible] the DB or MySQL or Postgres. NoSQL options like Mongo or Amazon allow for big data processing at relative affordable price. SQL relies on proprietary servers and storage systems which end up becoming more expensive than a gigabyte or transaction cost of NoSQL. NoSQL databases can scale out as opposed to scaling up and we said because the vertical and horizontal scale. In the past database had relied on scaling up by buying bigger centers to have more data loads. Of course by larger and larger servers. The problem is that ones the NoSQL goes down you have to reelect a new bus that will help with the right traffic. The problem with that is that now, your dataset that is now dot-dot accessible, it's actually even higher. If you get up, when he got bored of beef to the system, you actually give up some more availability. NoSQL can scale out bidding the database will be distributed across multiple servers as load increases. Therefore, NoSQL databases are usually designed with low-cost commodity hardware in mind. You don't really need specialized counter for many of these systems.

>> We briefly discussed about the NoSQL databases. Now, let's go into little detail about what are the flavors of NoSQL databases that exist. As we said, the NoSQL is a database that provides a mechanism for storage and retrieval of data. That is modeled in means other than tabular relations used in relational databases. The effective flavors you can see is like a key-value store, document, and graph, an in-memory data store, even a search. There is difference of these flavors. Let's go into key-value stores and some examples of them is like Redis, Azure Table Storage, Amazon DB, [inaudible] Riak are some of the examples. Stores data in buckets as key and a value pair. Each key must be unique and values don't have a type, they can contain any value at all. You can think about it as a hash table or a distributed hash table. This is the most common format. The ability of that is it provides very fast write operations. As long as you know the key, you could actually look at the data. You can actually write the data replace the data very fast rates if you select based on the key. For those who don't know actually time complexity, this can be like an O1 for a symbol, a key. But it can be very complicated when you try to search across the values because it needs to do a scan across different keys. It works great when you have a flat data model and schemas you can't model with RDBMS. It is considered some dictionary collections, hash table collections, and INI files, have a similar format with Key-Value Stores. A column store is like Cassandra, HBase, Hadoop, or even Bigtable. Data is stored as columns. Columns consist of a key-value pair and often a timestamp. Columns are grouped into rows or column families. Columns are grouped generally in tables. But there are no joins that are permitted. You still have some format [inaudible] tabular format, but then you still have it in a key-value system. It doesn't have a strict schema. For example, as you can see here below, you can have these ID and then you have a user ID. But in this ID you're going to have an in-reply in a different ID. No strict schema, its blazing fast processing abilities, it benefits seen through decentralization and scalability, and it doesn't allow joins between tables as in a relational system. The third one is the graph, like Neo4j and JanusGraph. Think of mathematics as edges and vertices, for example. The question here is; how did you really depict all that within a NoSQL system? It is composed of nodes and relationships on edges, both can have key-value collections and label tags can be added as well. It is ideal for where the data you have is defined by a relationship like for example, friend tracking on Facebook or movie database, or searching for things like friend of a friend of a friend who's not friends with me, or people who worked on a movie with me who haven't worked with my friend, or finding the shortest path between me and this person. Fundamentally speaking, with a graph database, it's a kind of work rate where the level of testing is very deep. But it can also be a double-edged sword because it can be really slow. The final flavor and I would say the most popular is the document one. Information is stored as documents like a [inaudible] , like a JavaScript Object Notation, JSON, or by Binary forms of JSON, or even XML. Documents are still collections of key-value pairs. But documents are collected in collections. The values can be like a Boolean or Integer or even different types of some documents or arrays, so you can detect them and even have different levels of testing. It's really fast of writing, it's flexible indexing for fast value searching, it provides for horizontal scaling, schema-free, schema-less. It is great as an object store as well.

>> One of the main things that somebody working with databases they have to do is what we call data modelling. A lot of things have been said about relational data modeling. It's complicated it requires some domain expertise to do that. As more and more systems move to the Cloud then the NoSQL database becomes more of a de facto thing, then I think that's what we'll discuss today. Fundamentally, the NoSQL data focus around what we said before as flexible schema, which unlike SQL databases where you must determine and declare a table schema before inserting the data. In this case you don't have to do that. This flexibility facilitates the mapping of documents or entity or an object. Each document can match the data fields of the represented entity, even if the data has substantial variation. In practice however, the document in a collection share a similar structure. What you would see as we move forward towards the slide deck is that we will focus mainly on the documents data just because it's the prevailing NoSQL paradigm, as well as because it's similar to explain. What are the challenges in data modelling? The key challenge in data modelling is balancing the needs of the application, that is the performance characteristics of the database engine and the data retrieval patterns. These are the three main things that we need to balance out. The needs of the application that performs correctly with database and the data retrieval patterns. Which means that if you can decide a random data model, without that I would say the application itself or the database or the retrieval patterns. When designing data models always consider application usage of the data, like the queries, updates and processing of the data as well as the inherent structure of the data itself. For example, one of the common things that you need to pay attention is, what types, whether this is like a point queries or what it's like scans across many keys, and what type of scans it may be, and then decide accordingly with the data model. Make a decision designing the data models for results around the structure of documents and how the application represent relationships between data. In the SQL space, automatically that can be done through joints, but this is not allowed or they should not actually allow those SQL because you don't have to wait. There are tools that allow applications to represent these relationships. One is called the references and the other is called the embedded documents. Embedded documents capture relationships between data by storing related data in a single document structure. In NoSQL, documents may make it possible to embed document structures as sub-documents in a field or array within a document. In this case, what you can see is you can have the contact and then you can have an embedded sub-document which is in the phone and the email, and then the access, then the level of the group itself. In this case, what you can see these now you have established some form of relationships between the data and you have done that through an embedded process. Embedded documents capture relationships between data by storing data in a single document structure. When you use embedded documents, you have "contains" the relationships between entities one-to-one relationships, and you can also have one-to-many relationships between entities. In these relationships the embedding or sub-documents always appear with or within view to the cortex of the one or parent documents. If we go back to this example, this is the sub-documents. This is a one-to-many relationship. Embedding provides better performance for read operations as well as the ability to request and retrieve related data to a single database operation. Again, if we go back and request the phone, we can say, db.IDX, find contact.phone, and within a single query you can actually tear back the sub-documents. Embedded data models make it possible to update related data in a single atomic write operation. Similar to the [inaudible] , you can also do an update of the data, and you can do that in a single operation. The second way you can do relationships is through references. References store the relationships between data by including links or references from one document to another. Application can resolve these references to access the related data. In this example, you have like a user ID,id object ID. Then you have a contact document and then an access document. What you can see here in both cases, we have replicated the ID. The ID field is different. But then you need to have a user ID, which actually points back to the previous document. It's subsets. This is kind of emulates adjoined within a relational database. When to use references? In general, use normalized data models: when embedding would result in duplication of data but would not provide sufficient read performance advantages to outweigh the implications of the duplication. To represent more complex, many-to-many relationships. To model large hierarchical datasets. Fundamentally, if you think about it right, embedding is where you have one to many or many to one and references when you have many to many relationships. Also when embedding will result in duplication of data. If you have, let's say, many times in the same field that you probably want to use references because then you can actually have one instance of the data. References provides more flexibility than embedding. However, client-side applications must issue follow-up queries to resolve the references. In other words, normalized data models can require more round trips to the server. Now these actually cause a number of issues because if you would update both your object, both documents at the same time and you cannot do transactions, then it may actually become the in causes to the state. Write operations are atomic at the document level, and no single write operation can atomically affect more than one document or more than one collection. This is what I just mentioned. If in the case of references, if you actually update your document, you could also have an atomic operation within that document. You cannot have an atomic operation

across different documents. A de-normalized data model with embedded data combines all related data for represented entity in a single document. This facilitates atomic write operations since a single write operation can insert or update the data for an entity. Just mention that. Normalizing the data would split the data across multiple collections and will require multiple write operations that are not atomic collectively. Let's go ahead and see a little more details about the query processing in the NoSQL space. NoSQL has some fundamental limitations that we need to be aware of. It calls for a more relaxed data consistency model. It provides primitive querying and searching capability. Obviously by data consistency we mean that now you don't have like consistent partition-tolerance. In some cases you have to give this up for [inaudible] available there, partition-tolerance or some other data structure. Many of the NoSQL DBs today is based on the Distributed Hash Table, which provides a hashtable access semantics. To access or modify any object data, the client is required to supply the primary key of the object, then the database will look up the object using an equality match to the supplied key. In all cases we said you provide the object ID or the document ID, or the key in the key value pairs, and then you return back the data as the value of the information. For example, if we use a Distributed Hash table to implement a customer DB, database, we can choose the customer ID as the key. Then we can or set or operate on any customer object if we know it's an ID. For example, if we have cust data, this will be a distributed hash table. Don't get over the customer ID. Then if we go to set some data, we'll use the same customer ID, then we will modify the customer data. Then if we work through some other query like execute we'll use still the customer ID and then the function that we want to perform with the data. In the real world, we may want to search data based on other attributes than its primary key. We may use also search attributes based on, let's say, greater or less than relationship. You know, this is going to code. Or we want to combine multiple search criteria using a boolean expression. You also have these leads. Some of the NoSQL databases provides an indexing and query processing mechanism within the local database. In this case, we can have the query processor broadcast the query to every Distributed Hash table where a local search will be conducted with result sent back to the query processor which aggregates into a single response. In this case, for example, you have the query processor which receives the queries, then it propagates the query across different nodes and receives the response and then responds back to the customer in every single response. This is some way some database actually work like that. This is some aggregation of the results and that's why these scatter and gather local sets.

>> Let us discuss today about the active cloud storage. By active, I mean the cloud storage that can be easily accessible by different types of virtual machines. The infrastructure as a service providers supports different types of storage, and hardware storage believe they provide storage for the users. One of them is the ephemeral storage, the second is the block storage, the third is the object storage, at the last one is the data archiving. Of course, there are other elements that they provide like file systems social service and a few occasion like caches of data before they actually get to the storage elements. But fundamentally, these are the four elements of storage. One important aspect here is that in many cases, object storage and data archiving are effectively in the same category [NOISE]. It depends on the storage access you have, like the block. A block is a fixed-size block in a specific location. The way you expose the block to the instances is through a specific file system. The second way you can access the data is through files. This is a specific folder or a file structure. When you have the location and also has the properties. The final one is the object, where you have a specific data and metadata that you have collected. Usually you need some form of APIs, like the data that use it as POSIX semantics as you would use in a file system. Ephemeral storage is effectively the old instance storage. Ephemeral storage is used to store data in the compute instance. The lifetime of the data are based on the lifetime of the instance, which means that when the instance goes down or when the data needs to be replaced, the data would also be used. An example is if we deploy a platform as a service data store like a database, we need to have a logical for data replication. An easy way to have that is because if the instance goes down, we may lose the data. An infrastructure as a service provider provides an instance family that include high storage instances. That means sometimes very fast SSD-backed instance storage, optimized for high random IO, provide high IOPS at a low cost. An example is i3's, i4, i5 families right now. The use cases could be like a NoSQL databases like Cassandra, MongoDB, scale out transactional databases, data warehousing, Hadoop, or even a cluster file systems. All of these we actually use [NOISE] their data storage because it's fast and because it's local axis. [NOISE] Another way that you could access data is through a block storage, and the block storage services allows the user to create storage volumes and attach them to VMs. From the operating system perspective, whether this is block storage or even the local storage, it actually appears as the same. Once the volume is attached, the user can create a file system on top of these volumes, ran a database, or use them in other ways. The fundamental aspect here is when you attach a block storage, you probably need the file system because users have been accustomed to files and folders format. Block storage volumes by most Cloud vendors tend to be placed at specific availability zones where there are automatically replicated to protect you from the failure of a single component. When specificity means that some of the high availability aspects of the block storage have now been taken care by the provider who provides the block storage. In comparison to, let's say, using the oldest storage, in which case, the user is now responsible for the replication of the data and make sure that there's high availability. In the block storage space, the vendor is now providing a highly available ecosystem. The block storage volume types offers durable snapshot capabilities that are designed for at least five dots of availability. This means that, [NOISE] for example, the block storage service can provide snapshots to allow that more durable storage that may provide durability. Then for that, when you come back, you cannot actually see the data. You don't really use date. The block storage system balances the creation, attaching and detaching to the block devices to servers. Usually this is done either through SAP for Buffett API or it could come through the CLI or data. You got instance you buy backs up like local interface callbacks and you try to attach the volume itself. [NOISE] Block storage is appropriate for performance sensitive scenarios such as database storage, expandable file systems, or providing a server with access to raw block level storage. When you think about the difference between the local storage, in most cases it is a network hub. The blocks storage case, because you attach the volumes, now you need to go over the network. However, by the Cloud providers, they provide specialized high-throughput interfaces only for the block storage. So you actually get really high throughput for the network. But then the latency is what matters from that point thereafter. The user can have different factors of the block storage. They can dynamically increase the capacity. They could potentially tune the performance, or even change the type of live volumes with no downtime or even performance impact. Data flexibility does not exist on a [inaudible] storage because of the instance, you cannot dynamically change the storage of the instance, but with blocks storage, you can say, "I need more space," so you can dynamically [NOISE] add more capacity. Signal with the snapshot management, maybe they provide this functionality of backing up data stored on block storage volumes. Network stops can be restored or used to create new block storage volumes. For example, you can snapshot something, say, "I'm going to create a unit stop and I'm going to copy the data that I snapshot that." That makes a lot of flexibility there for applications. You access these stops, you create another idea and you just need to copy the data. You don't really need copy because the provider provides the snapshot and restore mechanism to do that. [NOISE] The most famous block storage system which the AWS, EPS, [NOISE] there are different varieties they provide. They provide varieties where the regular solid-state drives and even hard drives. At the same time different performance and also cost will meet you

out of them. So in the volume type, there is what we call EBS Provisioned IOPS SSD, and EBS General Purpose SSD. The provisioned IOPS means that you actually get specialized IOPS to the disk. Effectively, your operations now have a low bow that they provide these SLAs. In some cases, it is the highest performance SSD volume designed for latency-sensitive transactional workloads, whereas the gp2 is focused on the general purpose SSD volume that balances price performance for a wide variety of transactional workloads. Now, sometimes you may want to use EBS with hard drives like st1 or sc1. These are usually low cost. The access and the throughput are not that important, but you want this flexibility of scaling up and down your storage as you attached it to the instance. You still need to work to use these file semantics. [NOISE] If you see a use cases, one use case would be for very high [inaudible] like those SQL relational database like you do thousands of queries per second, whereas it cope HDD, the file drives for block storage, this is when you have enough few scars per day, a few complicated operations. At the same time the data varies, but fundamentally, when you go down to the throughput, you can see the difference. One is obviously like four types backs throughput compared to the SUR, the IOR is four times larger. In terms of max IOPS, the IOR Cadillac batch faster. What it means is you don't only get the ability to attach blocks devices to instances, but you also get a variety of SLAs across these different services. [NOISE] In EBS, and this holds for many Cloud providers, at no additional charge for the customer, Amazon EBS volume data is replicated across multiple servers and availability zones to prevent the loss of data from the failure of any single component. That's the beauty. With the fibula storage, they usually has partner that with EBS, the provider provides that. Your EBS volumes are designed for a general failure rate of about 0.1-0.2 percent. Failure refers to a complete or partial loss of the volume depending on the size or the performance of the volume. Of course, when we think about the equivalent of, what if we used our old hard drives, this is about 20 types more that our typical commodity disk drives, AFR is about 4 percent. For example, you have about 1,000 EBS volumes for one year, you should expect one or two will have a failure. I get the failures are handled by the provider, not by the user, but this is the maximum you should expect. You hope this might be some service disruption, but it is low, because the provider provides them. EBS also supports a snapshot feature which we discussed in case you wanted to do some backup. If something fails, you still have a backup somewhere else. [NOISE].

>> Long-term storage, obviously the terminology itself may not be the right one but it assumes that you need to store a lot of data and you probably need to store it for a long time. You don't have very frequent access patterns [NOISE]. The fundamental component, this long-term storage area, is the object storage that is redundant scalable object storage using clusters of standardized servers capable of storing petabytes or even exabytes of data. Object storage is not a traditional file system, but rather a distributed storage system for static data such as virtual machine backups, photo storage, e-mail storage, backups, and archives. Mainly it is accessed through APIs. It has no central brain or master point of control. It provides greater scalability, redundancy and durability. Again, this is not about either [inaudible] files, this is mostly about using the application and services that need to access the data. [NOISE] Objects and files are written to multiple disk drives spread throughout servers in the data center and all that is being handled by the infrastructures service provider. IaaS software responsible for ensuring data replication and integrity across the whole cluster so again the user does really need to know all the details of how this is being done. Storage clusters scale horizontally simply by adding new servers. Should a server or hard drive fail, the IaaS vendor replicates its content from other active nodes to new locations in the cluster. IaaS vendors use software logic to ensure data replication and distribution across different devices, inexpensive commodity hard drives or servers can be used in lieu of more expensive equipment. Technically, the service provider deals with how the data be replicated and what storage system it is actually stored. Most of the Cloud providers either AWS or Google or rather, usually talk about eleven dynasty's durability and scale past tens of trillions of objects. The difference between blocks and objects is in one case you access the data through an API, in the other case you use this form of blocks. Obviously, as we said, block storage is when you tag that delivery to a machine whereas in object storage you use an API to access the data [NOISE]. Let's see now how do you really make a storage decision beginning with what we discussed, the ephemeral storage, block storage and on object storage. In ephemeral storage, you run an operating system and scratch space. In block storage, you add additional persistent storage to the virtual machine. In object storage, you store data including VM images, or even PDF files and so forth. In the first case, ephemeral storage works through a file system, in the second through a block device, but they subtract that through the file system. The object storage is through APIs or vendor-specific APIs. When we see where it is accessible, ephemeral storage that's the hardware of the VM, is actually within the VM, looks as we feed a VM, but it can be sent across VMs. Object storage is anywhere. Now, in ephemeral storage, the VM is terminated, so you lose the data. In block storage deleted by the user, the data, and again the object storage the data is deleted by the user [NOISE]. Usually, when we discuss about it or how about [inaudible] to add terabyte. It's a good space to store in ephemeral storage, but above that, you probably want to use some form of block storage or even store it in object storage because it can store up to exabytes of data [NOISE]. Now, S3 as the beach or AWS product that offers object storage, it provides different types of DSS similar for others, like Google, the airline, and so forth. It provides S3 Standard, S3 Intelligent-Tiering, S3 Infrequent Access and all of these come with different price tags. For example [NOISE], in S3 Standard you pay a lot for storing the data, and you pay very little to retrieve the data, but if you go down to S3 Glacier Deep Archive, that's what you actually pay a lot for the retrieval, but you don't pay a lot to store the data. Let's move to the next slide we'll see this, so when you try to see the different areas, like where each one is being used. All of them provide the same durability. The common use case for S3 Standard is when you have active data, whereas a Glacier you have a near-line archive. I would really [inaudible] archive data [NOISE]. As you can see, the list of the features that the provider provides, it's pretty much the same across all this. But with both dark is the slide right there, public pricing for S3 standard. There she's like Glacier, you see the storage per gigabyte is much more, it's multiples, it's an order of much more than S3 Standard compared to Glacier Deep Archive. Whereas when you see the retrieval line [NOISE], it's the difference. Is that the retrieval for Deep Archive is supposed to be almost two times more per gigabyte more than what is paid S3 Standard and the reason for that is Deep Archive is used for archival pitch, so a database, it's like to motivate you from accessing the data from the archive. Then there are different types of price tags in different perspective like if you add the data, the added data to S3 Standard is very low, but if you add the data, in Glacier, it's much more expensive. The rest of the course catalog with that. Common media and entertainment use cases, daily production, news, sports, web or social broadcast can use an S3 Standard, but when you need to really keep your code that preservation or long-term storage that when you actually use the Deep Archive. The use cases is different per product here. Let's now go to data archiving. We said here [NOISE], back in the days, Glacier was actually used to be separate at all that at some point it was embedded within S3 so that's why it's flat. But it barely providers provides this data archiving it down. They say, combined with those exponents to make it seamless out of it. IaaS provides a secure, durable, extremely low-cost cloud storage service for data archiving and long-term backup. Customers can reliably store large amounts of data for less than one cent per gigabyte. Actually said that it's actually worth less than one cent its much less than one cent per gigabyte. To keep costs low, low data character codes are optimized for infrequently accessed data where a retrieval

time for several hours is suitable as we say, like for example, Deep Archive can take up to 40 [inaudible] with backup data. Whereas in S3 Standard it can be really fast [NOISE]. It can be very useful to decrease the cost of storing data that you don't really access. You just store it there for, let's say once a year you may access it. Now, the other aspect is there's also other ways that you get to store as the attachment data, and one of them is, this is the device that AWS provides where you actually can use a USB to store types of data that you get translated to a centralized location and you can transfer like terabytes or petabytes of data. Actually, there was a law conference, they brought a whole track where you could actually use it to store a lot of the data, so AWS provides more and more and even other providers provide more and more ways that you can actually store the data. This is actually called the snowball device, that's how AWS calls it [NOISE]. As we said, there are different offerings that exist, based on cloud providers, Openstack and the blocks are received that addresses Elastic Block Storage. In block storage, Openstack Swift, AWS S3, Azure Blob Storage, Google use Cloud Storage. Data archiving in the AWS Glacier. Others, [inaudible] and so forth. Huge data transfer, AWS snowball, Azure Import-Export. You know different providers provide different names for each of these offerings.

>> Data pipeline without a messaging system. Let's start with this concept. Message in the Cloud. Messaging deals with sending data over the internet to scale applications. It is stateless, i.e., does not store data. It is used to synchronize Stateful applications. It basically carries the state but itself is stateless. The question is now, how are we able to send messages within the cloud? What released here is effectively some of the properties that those messages will have. Message broker is an intermediary program module that translates a message from the formal messaging protocol of the sender to the formal messaging protocol of the receiver. Some examples of some of the messaging broker is: Apache Kafka, Apache ActiveMQ, RabbitMQ, Celery Task Queue, et cetera. Messaging is an important piece of infrastructure for moving data between systems. To see why, let's look at a data pipeline without a messaging system. Before I go to this example, I want to mention that a lot of these systems are currently used in productions by many companies. This is the fact way of how you communicate between different types of services. They provide some added guarantees as well, that we'll discuss later on, like availability, and so forth. Let's assume that we have no messaging system. The system starts with Hadoop for storage and data processing. Hadoop isn't very useful without data, so the first stage is to get the data inside Hadoop. So far it's not ideal, but you can still get a data to Hadoop. Unfortunately, in the real world, data exists on many systems in parallel. The sources of the data that exist in so many applications, for example, if you have built a service that deals with, let's say, Amazon, you have got so much data that you generate with your try to shop some products. Effectively, you want to send the data to a system, big data system to do some analytics. In the first place you have at least all the systems we'll have to interact with, Hadoop. Let's provide more clarity. The situation quickly becomes more complex, but it's ending with a system where multiple data systems are talking to one another over many channels. Each of these channels requires their own custom protocols and communication methods, and moving data between these systems becomes a full-time job for a team developer. What we see here is we have systems like DevOps Metrics Applications that a [inaudible]. All of these are actually sending messages with Hadoop. But then you would extract some of these messages for a job to do some fraud analytics, or to do some monitoring, or to invest in some of the security measures. As you can see here, Hadoop now becomes a central system of injecting a certain data, whereas it was developed for specifically to big data. Now data effectively Hadoop becomes a data pipeline itself, which was not what it was created for. What will happen eventually is, you have a single system and all these channels and you end up in this case. When you have substantial congested roads because your system cannot hold the workload. The question is, how can you achieve some congestion avoidance [inaudible] control all this traffic that goes through your big data systems. [NOISE]

>> Let's discuss about data pipeline with messaging system. In the previous section, we discussed about data pipeline without a messaging system. This is about data pipeline now with messaging system. Let's look at this picture again. Let's assume that we use a messaging system. In this case, we'll assume we use Kafka which is one of the most common messaging buses that exist in the industry. Of course, one may assume that we knew something else, but for the sake of this example, we're going to use Kafka. What we see is that all incoming data from all applications and services are writing data first to Kafka as a queue. What happens is all the consumers, that is the Hadoop, security, data warehouses, monitoring, or even different services are actually consuming the data out of the queue. In this example, what happens is now the communication between services becomes more natural and everything goes through the Satellite queuing system or messaging system. In this model, it's actually what we call a pub-sub model. When we have a number of publishers that they are publishing. let's say some letters, and then they go through the channel, through a stream, and then they're been delivered to subscribers. We can easily see the equivalence of that with how, for example, our postal system is working where people are putting the letters in folders, and then the folders are being delivered to a USPS or to private companies like UPS and FedEx. Then they are delivered to the subscribers. Messaging consists of effectively the producers, which are the ones that are generating the messages. The consumers which are the ones that consume the message generated by the producer and they consume the message out of queueing system or out of the messaging system. Now, between the producer and the consumer, there is a message broker, as we discussed, that is used to synchronize the messages between them. Of course what may happen is you may have producers that produce a message, and then there you may have many consumers of that message instead of just one and vice versa. You may have multiple producers and less consumers. The message broker performs the synchronization of messages with the help of queues which are used to add new messages or remove messages which are consumed. Mainly what is the backbone now discussing is what we call the data in transit. That is how services on the cloud are able to move messages and to move information between different types of services that need to consume those messages. In the messaging, there is this part which called the store and forward. The broker implements the store and forward mechanism to synchronize messages. The queue in the broker will store the messages until and unless some consumer consumes those messages. The consumer consumes the message process it and sends an acknowledgement to the message broker to indicate the successful delivery of the message. This is effectively, producer puts a message to the queue consumed by the queue then acknowledges that it has received effectively the message. In some sense, this is at least somebody or at least one consumer has acknowledged the message. In decoupling. Now the producers and consumers that are separate from each other and they involve decoupling to exchange messages with each other. In the decoupling aspect is that effectively, what we have achieved, is we have achieved that the complexity of the code that we build in the consumer and in the producer are totally separate. You don't have to effectively block one with another, because there's a centralized queuing system that can hold messages in some cases for hours or days. There are three types of decoupling, the Logical decoupling, the producers or consumers do not know anything about each other except the routing information, and this is actually really great. The physical decoupling in which the producer and consumers are even or different locations. Finally the temporal decoupling, the messages are consumed by the consumer at a later stage. As you can see in this we have achieved logical, physical, temporal is you have this separation of the consumer and the producer. This makes much easier the maintenance of the service. For example, if you didn't have a queuing system and you published a message and it was stuck, then it will affect some of the CPU cycles of the producer. Now because you have decoupled the consumer and the producer itself, now you're able to effectively isolate that. You can also isolate the failure debates in physical decoupling because the producer and consumer are even located in different locations. Temporal decoupling is also great because you can publish a message and you can say, "Oh the consumer is going to consume that after 24 hours or after 48 hours", where maybe a bunch of other messages have also taken place. In this case, we have the queues, which are of course, loosely coupled as we said. For controller A, the controller B, and controller C. The accuracy if you have a sequence of microservices talking to each other, we may have some blockage situation, whereas in this case, we'll have to have queues. You have a queue that tries the controller and consumes the message, put the message back to another queue, consumed by another controller and another queue. Effectively, we can have a system that can maintain a queue of messages without blocking the functionality of any of the controllers themselves. This is effectively loosely coupling.

>> Let us see now one of the main messaging systems that is provided by one of the Cloud vendor that is AWS and one of them is SQS. There are a few other ones like Kinesis, there is Kafka, RabbitMQ, there's a lot of them. But let's focus on this example now, and then we can do another few other examples later as well. Amazon Simple Queue Service, SQS is fast, scalable, and managed. Fast means that it can scale. So it can consume really fast some of the messages and managed means that AWS is taking care of the management of SQS. It enables asynchronous message-based communication between distributed components of an application. We discussed earlier about coupling of the set of system with cubic systems. So what can it do? SQS can decouple the components of a cloud application between the producers and the consumers. So you may have, producers producing messages that are being routed to the queue or multiple queues, and then consumers are contributing messages out of the queues. SQS can be used to transmit high volume of data without losing messages and does not require other services to be available. That means that if consumer is down, it can still keep the data within the queue. Once the consumer is up, then it can send the data to the consumer itself. These are example of an SQS where you have a Producer, that produces the data in a Master Queue. Then you have a Master Queue Order and then you have the consumers, which can be, for example, S3Q or Elastic Search Q or RDSQ. So what you do here is you fully decoupled the producers with the datastores that the data needs to be. If the data need to be let's say in a object store like S3, it's fully decoupled but will the data be consumed, let's say by Elastic Search or will it be consumed by relational database system like RDS. You also may build independent queuing systems within those subsets of systems as well. So this is an example use of Amazon SQS. You know, some of the major features that the SQS provides, and I'm confident that many queueing services also provide, is it provides the data infrastructure. So if something goes bad, you still are able to right with the queue. It provides multiple writers and readers. The reason you do that is because you need to have the ability to have multiple producers and multiple consumers but as well within a producer you got the issue that you could actually write the queue in a very fast, multi-threaded way. You can configure the setting per queue. That means you could configure the size of the key, and of course, as you can imagine the more data are being saved within the queue, the more you have to pay to AWS to host that SQS. At the same time though, it may be the case that you may want to actually hold 48 hours data before you write the through and many types have of these queuing services are used to generate financial reports. So you may want to wait until you have all the data but the same time you don't want to lose the data, so you write them to the queue. It's highly available, persistent and then its being consumed, that's my point. The messages can vary but, one of the issues that we have seen at least with a few users of SQS is that it has a message size of up to 256 kilobytes. That means your message size have to be fairly small, right. It provides access control, which means that now you can configure the security mechanism. This is actually part of the beautiful things about AWS, it provides integrations with many other services within the ecosystem. So access control becomes like the factor based on other services and it can also use delayed queues. So you can say publish this message after a specific amount of time, so this is a great feature actually. After a message is consumed from the queue, SQS does not delete the message. That is because the system is distributed, there is no guarantee that the component will actually receive the message and obviously SQS does not leave the message, and instead, your consuming component must delete the message from the queue after receiving and processing it. The reason you do that is because you want to decouple the consumer from the producers and at the same time you may want to have maybe multiple people that want to consume the message. So you may have multiple consumers. So if you have multiple consumers, and then one of them takes the lead to actually delete the message after it does some sort of form of coordination. So there's also something we call message visibility timeout. That means the amount of time that you can keep the message before it actually becomes non-visible. So we don't want all the components in the system receiving and processing the message again. In some cases, Amazon SQS blocks them with visibility timeout. It is a period of time during SQS prevents other consuming components from receiving and processing the message. Again, this is a very useful feature because you can actually say, I have consumed the message but I don't want anybody else to consume the message for the next, let's say five or ten seconds. So this can be done. So, this is a great feature that SQS provides. So let's discuss about message lifecycle. Component 1 sends a message A to the queue. So the queue has it. Component who retrieves the message A from the queue and sends the visibility timeout, period starts. Component 2 processes message A and then deletes it from the queue during the visibility timeout period, so what happens here is that you consume the message, you process it, and once you're at 3, now you can delete the message. So the fact that you have the ability to do, what we call a visibility timeout, means that you don't need to build coordination across all the consumers. Because [inaudible] two consumer receives the message, maybe its stuck processing the message but the processing is now decoupled from the consuming, which is really great because nobody else can consume the message so you don't have duplicate messages across different services. If this is the requirement, because [inaudible] requirement is for multiple consumers to consume the same message as well. So it depends on what

you need. You can use the visibility timeout clock. You also have the ability to do what we call the delayed queues. Delayed queues allows you to postpone the delivery of the message in a queue for a specific number of seconds. For example, you can create a delayed queue, so any message that you send to that queue will be invisible to consumers for the duration of the delay period. So delayed queues are similar to visibility timeouts in that four features, make messages unavailable to consumers for a specific period of time. The difference, the difference between the two of them is that for delayed queues a message is hidden when it's first added to the queues. Whereas visibility timeouts, a message is hidden only when the message is retrieved from the queue. That means a delayed queue, you effectively delay the message to be visible within the queue. Whereas, in the visibility timeout you delay the message from being viewable to the consumers. So it basically is the same but it is different between each other because it affects the consumers versus the producers.

>> Let us cover the last section, the RabbitMQ and exchanges. RabbitMQ is a mode, the AMQP model contains producers, consumers, exchanges, queues, bindings and message brokers. We covered about producers and consumers. Let's discuss about the exchanges, we discuss about queues and we discuss about message brokers. Exchanges, function as putting the messages received from the producer into the first queue present in the message broker. Exchanges is effectively part of the whole ecosystem. Now, queues buffer up the messages, until the consumer decides to fetch the messages. The link between exchange and the queue is called binding. Effective right now we are defining, exchanges, queues and binding. Again, exchanges function as putting the messages from the producer into the right queue. Queues buffer up the messages, until the consumer decides to fetch the messages. Binding is the link between the exchanges and the queues. There are three types of exchanges, which effectively tell us how the messages will be routed from the exchange to the queue. The first, is the fan-out exchange. In the fan-out exchange every message received from the producer is stored in all the queues present in the message broker. In the direct exchange, the messages are stored in a queue when the routing key provided by the producers matches the binding key of the queue. In the topic exchange, it involves the use of wildcard keys in order to store messages to a particular queue. Fan-out, direct and topic are the main three exchanges. We put messages to the queue. Fan-out exchanges, as we said, every message received from the producer is stored in all the queues. As you can see here, we have a message x, now this message is sent both the q1 and q2 and effectively is consumed by the consumers of each of the queues. Effectively this is more like, broadcasting a message across the queues. In the direct exchange, the messages are stored in a queue when the routing key is provided by the producer matches the binding key of the queue. Effectively, the producer itself includes a routing key, and the routing key is equal to the binding key, which means that, for a message x with a specific routing key, the producer now says, this could go to q1, or this is going to go to q2. The producer now does what we call a direct allocation of the message within the queues. That's why it's called the direct exchange. In the topic queue, is where effectively the producer says, "I have a set of topics and then I'm going to add information [NOISE] for each of the topics." The topic could be, for example, anything starting by "shape." will be in q1, anything starting with "shape -" is going to q2 and anything with just "shape" is going to q3. Effectively you can substitute the topics with specific dates that's represented. That's why it allows you to have specific topics, specific tools for specific [NOISE]. The retrofit use is described by the following three methods. The first is the transfer. The transfer takes place it to the ends of the queue. This is two-sided, consistent order of basis within the queue. The second is the browser. The best is stored in the queue are browsed from the side of the queue. This ensures a high amount of parallelism who are asking the messages from the queue and also shows very high throughput synchronization. Whenever a copy of queue present a message broker is created. It is that using a particular order or using the sequence number. It is similar to how the TCPI works, like TCPI uses also some forms of sequence numbers. This is another example of like messaging in the Cloud. When you have producers and consumers, you have messages, and you have queue. Effectively you are a service that does production, adds messages to the q and then consumes at a later point of time, and the you will have sales that have more consumers and producers and so forth. The q becomes this defacto pipeline between the services. How do you handle failures? There can be failures in effectively the number of loads and huddle in q or effectively in the brokers or in exchanges or anywhere. Failure to receive any acknowledgement from the broker, the producer will try to set the message again. It will retry the message, which may lead to duplication of messages. There is a potential for prospect of duplicate messages where consumers fails to acknowledge or processed messages, which is also known as redelivery. What happens if you have duplicate messages either at the production or at the processing of messages. One may cause the producer to resend the messages and the other may cause the redelivery of the messages to the consumer. What is certain to know, is about, even if these values you might have set like at least one delivery at the any common failures. But it's really hard to guarantee effectively ordinate.

>> Today, we'll discuss about a messaging system which is called Apache Kafka. It's part of the open-source Apache foundation which most of the code is open-source and a lot of the documentation can also be found online. Kafka is a major component across the world in many industries, it is used by most companies right now because it provides a lot of great guarantees in terms of how the data is moved between different systems. Kafka comes from the German novelist Franz Kafka. Jay Kreps, who is one of the co-authors of Kafka, chose the name since it is a system atomized for writing. Kafka is a publish-subscribe messaging and it is rethought as a distributed log. I have posted a link here of the Kafka documentation incase you want to go a little bit deeper and I totally suggest that you would. Look, the data would not be lost that means it's highly available, so people were actually be able to write at all times. It is also resilient if the data gets lost and so forth. It does that by keeping multiple copies. In the real world data systems, these characteristics by Kafka are ideal fit for publication integration between coupon rates of large-scale data systems. An example that we provide here is where you have like the other basis, what do we have connectors that connected databases to the Kafka Cluster. It writes data to the Kafka Cluster and then what you do is you could also have applications, that write data to Kafka cluster. Then what you have is you have what we call stream processors like systems that actually takes up data, breachs data or takes data. An example would be, assuming you want to write the data for the actors. But you ought to enrich the data of the actors with some specific colleges they have that you have an application. You may save the data [inaudible] enrichment of the data that I maybe have like consumer applications. The basic architecture of Kafka is organized around a few key steps. The first one is that topics, that is all Kafka messages are organized in to topics. If you wish to send a message, you send it to a specific topic and if you wish to read a message, you read it from a specific topic. The second data is producers, you have the people who are actually all applications. What I'm actually producing the data. Producers effectively, what they do is they push messages it to a Kafka topic. Consumers on the other hand, is the applications that you're scripted data out of the Kafka Cluster. Now the brokers Kafka, as a distributed system, runs in a cluster. Each node in the cluster is called the Kafka broker. That means that you could actually have a broker of the cluster itself. Now let's go and see in more detail about the topics the brokers that consumers and the producer. Kafka topics are divided in partitions. Partitions allowing you to parallelize a topic by splitting the data into a particular topic across multiple brackets. Each partition can be placed on a separate machine to allow for multiple consumers to read from a topic in parallel and as we do that is both for availability as well as for performance. Consumers can also be parallelized so that consumers can read from multiple partitions in a topic allowing for very high message processing throughput. What you need here is you have partition was zero, one and two. You have the best is being posted in parallel so you have high availability but the same time you can have parallelization of how you consume the data from the producers. Each message within a partition has identified, it's called an offset. The offset is the ordering of the messages as an immutable sequence. Effectively we put that offset asset ID to make sure that we know where the data. For example, here, you can see also maybe 0,1,2,3,4 and so forth. Kafka maintains this message ordering so it provides ordering guarantees. Consumers can read messages starting from a specific offset and are allowed to read from any offset point they choose. I like consumers to join the class at any point in time as they see fit. It means that now you have this queue and now you have consumers that either data to Kafka and that means they can consume data from any point during the queue, they can consume data at any offset. Offset is used as an identifier to know where and what you have read. Given these constraints, each specific message in a Kafka cluster can be uniquely identified by a tuple consisting of the message's topic, the partition, and the offset within the partition. If we go back, this is the topic, this is the partition and there we have the offset. A topic has an ID, a partial ID. All of these elements is to identify the offset of what this message belongs to which topic, which partition and what offset it has. Let's go ahead to Kafka partition. Another way to view a partition is as a log. A data source writes messages to the log and one or more consumers reads from the log, and at the point in time they choose. In this case we have the log 0, 1, 2, 3, 4, 5, 6, 7. We have a data source that writes the data and then you have a destination that uses offset there at four and a destination that uses offset there at seven. In this diagram, a data source is writing to the log and consumers A and B are reading from the log at different offsets, which is perfectly fine. Now each holder holds a number of partitions and each of these partitions can be either a leader or a replica for a topic. Of course, a replica means that you have a liability. All the writes and reads to a topic go through the leader and the leader coordinates updating replicas with new data. If a leader fails, a replica takes over as the new leader. In this case you have leader partition 0, leader partition 1 and leader partition 2. They take the writes and the reads and they manage to the replicate the data across the board.

>> Now, discuss the introduction for Kafka. Let's now go and discuss about producers and consumers, and then we'll go a little deeper about consumer groups as well. Producers write to a single leader. This provides a means of load balancing production, so that each write can be serviced by separate broker and machine. In this case the client writes partition 0, and the data being replicated to the different brokers, like Broker 2 and Broker 3. The producers is writing to Partition 0 of the topic, and Partition 0 replicates that write to the available replicas. Associate leader is red, replica is blue. Now, let's assume that the client is writing to a different partition. Then that partition broker number 2, partition 1 will now replicate the writes to different replicas in blue in other red brokers. So irregardless of what the client writes to his broker, the data will be replicated across different brokers. Consumers read from any single partition allowing to scale throughput of message consumption. This is similar to what we said before in a master-slave database, where effectively the client writes the buster note, in this case, for Kafka, it is the leader, and then reads can happen from any partition, whether it's blue or red. Consumers can also be organized into consumer groups for a given topic. Each consumers within the group reads from a unique partition and the group as a whole consumes all messages from the entire topic. [NOISE] Now, let's see the possibilities of how consumers and partitions can be distributed. In the first case, if consumers is larger than partitions, consumers will be idle, because they have no partitions to read from. Now, if partitions are more than consumers, consumers will receive messages from multiple partitions. This is a good thing, right? Because you don't have any consumers being idle. Now if the consumers and the number of partitions are equal, each consumer reads messages in order from exactly one partition. So, you don't have neither consumers that are idle or partitions, that are idle. Let's see an example here. In this case we have Kafka Cluster. We have server 1 and two partitions, zero and three. We have clustered server 2, which has two partitions one and two. We have two consumers group, in this case, consumer group A, and consumer group B. Consumer group A has two consumers, C1 and C2, whereas consumer group B has four, consumers 3, 4, 5, and 6. As we see here, in consumer A, now you have consumer number 1 consuming from two different partitions that is because the number of consumers is less than the number of partitions. [NOISE] Whereas, in consumer B, you have equal number of partitions and equal number of consumers within the group itself. Now, let's move to a few more interesting topics. We have discussed earlier in this course about distributed systems, consistency, availability, and partition tolerance. Let's go through our assumption. Assumption is producing to one partition and consuming for one partition. This does not hold when you're reading it from the same partition, using two consumers or at the same partition used to produce, as we said before. So, Kafka makes the following guarantees. Message sent to a topic partition will be appended to the commit log in the order they are sent. That means, within a partition, you can keep ordering. A single consumer instance will see messages in the order they appear in the log. Once you consume the data from a single consumer, you will see exactly the same number as it is in the partition. A message is committed when all in sync replicas have applied it to the log. That means that once you write data to one of the leaders. If the data has been committed to all the replicas then you can say that this has been applied to the log. Any committed messages will not be lost, as long as at least one in sync replica is alive. So this means that if all sync replicas have acknowledged the write from the leader, that means that if the two dots go down, still the last replica can still serve the data to the consumers. [NOISE] So the one and two guaranteed ensure that message ordering is preserved for each partition. Note that message ordering for the entire topic is not guaranteed. What we see is within a partition, actual data, you definitely will have more of it. But across partitions, you may not. Now, the three of the four, the commitment type of data guarantee ensures that committed messages can be retrieved. In Kafka, the partition that is elected the leader is responsible for synching any messages received to replicas. That means that the leader would get the right and make sure that all the replicas are in sync. Once a replica has acknowledged the message, that replica is considered to be in synch with the leader. If it does not acknowledge that means its not in sync. If we go back here, let's see. This is the partition he tried to replicate. Partition 1 and partition 1. If both replicas have actually acknowledged that the write has been made that we consider all three in sync. Let's continue. That all messages are sent to the partition's leader. The leader now is responsible for writing the message to it's own in sync replica and once the message has been committed is responsible for propagating the message to additional replicas on different brokers. So we see it gets a data. Make sure that broker 2, broker 3 also receive the messages. Each replica, that is two and three, acknowledge that they have received the message and now can now be called in sync. By doing that, you can effectively have a multiple broker going down or going out. So let's go through the different failures. The writes will not, let's assume you have a replica failure. Writes will no longer reach the failed replica and it will no longer receive messages, failing further and further out of sync with the leader. So if this fails, this is out of sync. We see now second replica failure. The second replica will also no longer receive messages and too will become out of sync from the leader. At this point in time, the only replica, that is in sync, that is the leader itself. In Kafka terminology, we still have one in sync replica, even though that replica happens to be the leader of this partition. So this means that these are out. This is the

only available. Effectively, if the leader also dies, we're left with three dead replicas. Now replica 1 is actually still in sync because it received the write. But it cannot receive any new data. But it is in sync with everything that was possible to receive. Replica 2 is missing some data. Replica 3 is also missing some data. So, we still have the data replica 1 up to the point that it failed. But no new writes will be accepted. But the other half, partition 2 and 3, are out of scope because they have failed. So there are two possible solutions. The first scenario is to wait until the leader is back before continuing. Once the leader is back it will begin receiving and writing messages and as the replicas are brought back online, they will be made in sync with a leader. So what this means is, any new writes will fail, but once the leader comes back, it will start acknowledging the writes and it will be back in sync. Now the second scenario is to elect the first broker to come back up as the new leader. This broker will be out of sync with the existing leader and all data written between the time where this broker went down and when it was elected the leader will be lost. The other way around is effectively to wipe the whole state that existed in the commit log and effectively start from zero. So that's the two options you have right now in these failures. [NOISE]

>> Let's do a summary of the data streaming technologies that exist. One of the technologies that exist out there is RabbitMQ, and many times you'll hear that technology as well. Use Kafka when you have a fire hose of events right? It needs to be delivered to the partition order at least once, that means one consumer must mix at least on data. With a mix of online versus batch consumers, you won't be able to re-read the messages continue with current limitation that are around node level HA, high availability or you could even use like the trunk code that is posted online. Well, the other had RabbitMQ. It's another event streaming system. It's for lower number of messages, right? You need to be routed in complex ways to consumers, you want to prepare message delivery guarantees, you take care about all the delivery HA at the cluster node level. However, I'll have to say that Kafka has become the most popular technology overall in streaming. Especially because it's open source and many companies are actually using it. There's also other services like Amazon Kinesis. If you use AWS and even other Cloud providers, many of them actually they provide their own messaging system and it all streaming system. One of them is Kinesis, which comes in AWS. Some of the similar feature include like messaging system for large-scale real-time data processing, high-performance, highly scalable, low latency, fault tolerant. But there are differences. The other is like fully managed by AWS versus effectively Kafka, because you need to run it by yourself as a company. You need to trust the technically open source code or effectively you need to pay another common like Courtflow or something else that will host Kafka for you. However, there are many variations of Kafka that may exist in different cloud providers that you can also use. The second aspect is from Kinesis you actually have all these beautiful services the grazers like, for example, you could use DynamoDB, key-value store that effectively sends the data to Kinesis stream and that you can consume this data to do something else, so It's very well integrated. Whereas if you were to plug in like Kafka to DynamoDB, you have to build some of these technologies how you actually take data out of the 20 different queuing system. In the day and this goes with everything within AWS, all the services they provide, they try to make it very well integrated with each other. The reason is that the benefit of actually staying within the ecosystem is the fact that they provide you all the variety of services to get. Data durability and performance trade-offs, again like Kafka probably is better as an ecosystem itself, It's more durable it has better performance guarantees, It has like better limitations on the size of the best, as you can send. But I guess this balances out this on whether you are the man of service or to run it by yourself. I will propose to take a look at some of these websites that actually includes some of the information which I discussed in the section and actually the Kafka documentation is very important for you to understand this ecosystem. Thank you very much. [NOISE]

>> So let us discuss now microservices. What does microservice means in the cloud framework, and how do you move from a service-oriented architecture which was the initial phases of the cloud towards a smaller and tinier components of the services which we call the microservices. Most [inaudible] resulted in the cloud native framework where effect of data, most of the things on the cloud like Netflix, they deploy hundreds of microservices on a daily basis. When it needs to actually try to create a framework that can showcase the interaction between microservices, it becomes this very strange eye. This is, for example, a depiction of the Netflix services ecosystem and all the interaction happen between different types of microservices. So back in the old days in the generation of the services, there used to be what we call monolythics. Monolythics is effectively where you have a single process, where you have all the business logic. Then as you try to replicate the process what you do is you now create the multiple servers. So this is how you scale it. If one of the servers goes down, we still have the rest of the servers. You've already have some load-balancing in front of that. At the same time they didn't all decreases, you can distribute like that. A Microsoft architecture, with its element or functionality though in each separate service. The reason we do that is because we are usually the ownership of each of the services, as well as the logic so that the one does not affect the other. When we try to scale microservices, we effectively scale all the sub-components. So you may have actually servers that may run two instances of the same microservice or more than that in some cases, as we saw like in this example here, in this example here, and in this example here. The microservices architectural style is when you develop a single application as a suit of small services, each running in its own process and communicating with lightweight mechanisms. One of them is the Rest API, more bot and communications are based on what we call GRPC, like our remote framework of communication. What we see here is effectively the software development lifecycle, HPLC. Where effectively you build the system and then you sip it, and then you write and then you do this feedback loop around and improve the system while interacting with action. What you have is you have some source, where it has a deployment distraction when we start discussing about containers that may be like a Dockerfile, that a single file includes that, then you have development, you have QA and CI/CD, this is effectively automated the deployments. QA is where you do like testing of the code. Now you need to adapt to different types of effectively platforms that you deploy that. These can be like a physical hardware where you need some infrastructure management and then you need some virtual machines, then on top of that you have different logic. You become like a visually deployment where you effectively have a cloud provider provides that and still use the virtual machines. In the microservices style, you can also scale between different kinds of clouds, but it is like global, this is like Google cloud or IBM cloud, or different types of other cloud providers. Why do we want to move from monolythics to microservices? That's because microservices allows us for what we call fast deployment. So when we make a code change in application for each service can be made independently, as long as we keep the API contract with the other services are the same. That means that effectively the build test deploy cycles that speeds up dramatically and at the same time, you can make changes as long as you don't violate the context. For example, Netflix deploys code a hundred times a day, thanks to the early adoption of microservices. [inaudible] with facebook recently they where moved from a monolythics to a microservice style. It ask for efficient scaling, this has loads of a microservice can be scaled independently. This is a much more efficient way of scaling application, because not every part of the application experiences the same amount of load and needs to be scaled equally. It also provides design autonomy. Developers get the freedom to employ different technologies, frameworks, and design patterns to design and implement each microservice. So that means that you don't have a single framework like a single line bits, but now anybody can deployed in their own way. At the end of the day, the interactions, the contracts, they remain the same. So in some cases, you move fast, you fail fast, and your failure is actually very small, because the only fail your own microservice in the whole ecosystem.

>> So let's move on to the second part of the microservices where we're trying to introduce the containers. From microservices to container, Agile software development is a broadly adopted methodology in enterprises today. Effectively Agile is where you continuously iterate over the software we are building and deploying. So now I want to moving towards the idea of full-stack-responsibility for individual services. So back in the old days, what would happen is there will be somebody who will do the development, somebody who will do the quality assurance like test the codes, someone who will deploy that. So you will have these different verticals of people that, once you start deploying services on the Cloud, you have different types of externalities. But we've seen that this has slowed down a lot. So more than our companies what they apply is this full cycle responsibility where somebody is responsible for the beginning to the end. This is because we're moving, from a monolith where you don't have like centralized testing, but you have like isolated business logic testing only for the service that someone develops. Infrastructure needs to scale differently and the self-service model for project is taking center stage. So what we see here is we see different Cloud providers like Amazon, Google, Azure, Openstack, and so forth. Then you have, virtual machines and on top of that you deploy different types of microservices. Containers now are becoming what we call the foundation for this like along with proper DevOps and orchestration. So what's a container? A container is more like a lightweight VM, it's not really a VM, and we'll discuss it in a few seconds. So when we discuss about microservices, let's think about the complexities and the design considerations. We need to do have like a distribute application logic. With microservice, the logic now is spread across the services and more importantly, these embedded in the control and data flow between these services. You have now diverse technology stack. The system maybe comprised of in-house developed services, open source software and external libraries. So what that means is when you have microservices and you have these independent functionality, you can have a service that is based on robots or software and then you could have in-house developed software, as well as services. A great example in some of the companies, they may use like an open-source databases like Cassandra. Then they will develop their own way of accessing. The data Cassandra from it's own a specific side is developed in Java. Or it might be, they maybe using some sort of other inop the famous open-source project. One of them is like Envoy which approx developed by Lyft. At the same time at the backend, you may have something else that you have developed in-house. One of the issues now of course with microservices it's kind of hard to test and debug, when you have all this interactions, the first of interactions with the hundred microservices deployed, as you try to identify where is the problem within the stack, whenever you actually see an issue, it's kind of hard. Then sometimes we may get into the part where we will have what we call the butterfly effect, like a minor change of a service could eventually be catasrophic for the whole service. We have seen that in many cases. So many times, what happens is the focus of the whole company becomes about, how do we disengage in it's service? The errors, so it doesn't have a compounding effect in other services. An example would be if for example, a service is getting out of memory or is generate multiple traffic that the downstream services now have to accommodate that. So it creates more pressure, they ran out of memory and so forth. It can escalate too many services. So it can be catastrophic. So containers in some sense it is related to microservices, but it's not like a 100 percent about one matching. Container is more like an infrastructure. So this is more the new concept of virtualization solution for platform as a service and infrastructure service due to containers increase in density, isolation, elasticity, rapid provisioning. So what happens is when you have a VM. You need to effectively start a VM board in your operating system. So you need to take some amount of time to start a VM. It usually takes a few minutes to put a VM, check everything and so forth. With containers it's more like a lightweight package instead of a virtual machine. So you move effectively from a large monolithic application that has to be developed within a virtual machine, to smaller type of components that can be just deployed really fast with containers. So containers is actually [inaudible] top operating system. So you're going to actually slice and dice operating system. So at the same time, you can containerize parts of the applications. We're going to learn that through the homework as well. When effectively some part the application is one container, some other part is the other container. You can move parts of applications in different types of cloud infrastructure. With virtual machines when you produce a virtual machine you have a basic something that is related to a specific cloud infrastructure, whether it's AWS or Google or Microsoft and so forth. But once you move towards containers, now you can move containers between different operating and different Cloud providers as long as they have the same operating system. So if you use Ubuntu then you just need move containers across Ubuntu operating systems. It simplifies immigration complications between private, public and hybrid Clouds. The beauty about that is now the other frameworks like Kubernetes, where people use it to schedule containers across different Cloud providers. So let's see about, how did we come up with this idea of containers? If we see about the history of the cargo transport, back in the day, we will have all these goods. Cars, oil and a pianos and so forth. What we'll do is we'll try to do a mapping between that and the ways that we can transport the services in these goods. So that was how we use to effectively move data. Now in the last 40 years, we move towards a

solution where say, okay, the phase between the transport and the actual goods becomes container. So as you progress, you can't fit everything within a container. Now what the transport happens now in moving containers. Only at the last mile you're actually moving the actual products. So this is the same idea. So if we see the same analogy with virtualization. Back in 95, we had a thick client-server applications on a thick client, well-defined grand type operating system middleware. Then you run that in monolithic fiscal infrastructure. To those [inaudible] there after you know now the clients are becoming thinner and thinner you have applications that effectively just on mobile, just do rest APIs. Then you have services that are being developed in smaller elements, business logic. Then you have some sort of running on physical resources or Cloud or private infrastructure virtualized. So effectively what we did is we move from your millions of dollar of spending in many Frames towards people now using, spinning up by hundreds of virtual machines pretty easily to solve the problem. Of course, when we try to think about what will be the problem if we do that. Because when it is effectively we see like they separate former Soviet. Now you have not rerun again or on the physical hardware, but you have to run it on top of virtual machine sort of hypervisors, which means you are actually paying an overhead for that app performance overhead for that, because you're not directly accessing the memory. That happens through the hypervisor. So when we try to take the same issue with the container ecosystem we discussed before. You have to start a website, you have database, you have Web frontend, you have Queues, you have Analytics DB. So you have all these different elements, all these different services with your echo system. What we'll do right now we're moving towards an idea of deploying that of containers. The containers now can then be deployed anywhere. So it becomes oblivious to how you deploy these microservices to the actual hardware that you deploy the services. So now you can deploy the container on top of a QA server or develop the system on a public Cloud on a production cluster, or you can even develop it on your laptops. Now the great thing about that is now you can still have containers on your laptop, which actually gives you a significant advantage because the same [inaudible] and adopt the formula there are the same behavior that you see of the business logic on the container on your laptop will actually be the same as you move to the Cloud.

>> Let's go a little deeper in the containers. Why use containers? Obviously, once you deploy a container, you're built and deployed times decrease a lot. Because now you don't have to wait to spin up a virtual machine, which means that you don't have to wait to take some space for hard drive and allocate some memory, and all the computational intensive things that require to create a virtual machine. You can control the cost and smaller granularity because right now you're not really saying, "I'm going deploy on VM." But you have one VM and on top of that you can deploy 10 containers. You go at a much deeper granularity about how you can control the cost. Container technology simplifies cloud portability. So that means that you can run same application in different environments, which means that this is a big thing. With cloud providers, you're spinning up VMs. Now we're at a level higher, we're going into containers. If you have a container, you can move the containers between different cloud providers so they'll be between private and public clouds. It makes it very easy. Container encapsulates applications and define the interface with surrounding system. In a virtual machine, a full operating system installed with associated overhead of virtualized device drivers. This means that the virtual machine really need to get a specific [inaudible] system like an Ubuntu or CentOS, or even Windows. With a container, you only use part of this operating system. Container use and share the operating system and device driver of the host. Virtual machines have a full operating system with its own memory management. Containers share the host's operating system and therefore are much lighter. Let's go and see what's the difference between a virtual machine and a container. In a virtual machine, you have the visualization area or the server hardware. You can have the same container. You can have a host operating system. You can have a hypervisor, and on top of that you deploy the operating system. Like, you deploy another operating system on top of the host operating system. Then you have your binaries and libraries, and you have your application logic or your business logic here. Whereas in a container, what you have is you have the host operating system and you have some binaries and libraries that run on the same host operating system. On top of that, you have the application. So you effectively don't have a hypervisor layer which does a lot of translations. You directly [inaudible] slice and dice, the actual host operating system. Let's see what you can do in container. You can get an shell, you can own SSH in a container. You can own the process space. You can own the network interface. You can run as root. You can install packages and run a services. Let's see here what you cannot. You cannot use the host kernel that's because many containers use under the same host kernel. You cannot boot a different operating system. So your operating system and the kernel must be similar or the same along with what you deployed underneath that. You cannot have its own modules. You [inaudible] need the init the PID 1. Well, doesn't matter much for many people, but does matter for a few. Now, what's the difference between a container and a virtual machine? Let's see now the difference between a container and a process. Containers are processes with their full environment. A computer science textbook will define a process having its own address space, program, CPU state, and process table entry. The program text is actually memory mapped from the file system into the process address space and often consists of dozens of shared libraries in addition to the program itself, thus all these files are really part of the process. What we see here is, containers are processes in some sense. In this case in a container, same as every process, you take pieces of the memory and you tag them along. You take some other libraries and folders, and you actually put them in a container. How did containers start? Containers [inaudible] from the Linux container. It's a library called LXC, operating system level virtualization method from Linux. LXC is an OS virtual for running estimated Linux systems, container on a single control host. You have the Linux kernel which is a single- [inaudible] system. Then you have the kernel API which actually balances the containers, and then you have container management tools, which manages the high-level elements of the container. Within a container, you have different process and different name spaces. So effectively, this is the host operating system. These are the containers and these are the processes that you can write inside the containers. What do you have with elaborate system is the container management tools, which actually it's an application, it's not even within the operating system. Then the kernel API that the container management tool uses in order to leverage the containers. [NOISE]

>> So fundamentals of cgroups. So the question that we're trying to run is, yes, container is, has been a recent trend within the cloud computing, but is it really a new thing? Cgroups, control groups, is a mechanism to control resources per hierarchical groups of processes. Cgroups, abbreviated from control groups, is a Linux kernel feature to limit, account and isolate resource usage of process groups. This work was started by engineers at Google 2006 under the name of process containers. In late 2007, it was renamed Control Groups due to the confusion caused by multiple meanings of the term container in the Linux kernel. It merged within the kernel version of Linux in 2.6.24. Since then, new features and controllers have been added. So what we see is that effectively the technology to enable containers has existed for about 15 years now. So cgroups design goals. The goals was to provide a unified interface to many different use cases, from controlling a single process to whole operating system level virtualization. Control groups provides resource limitation. That is, groups can be set not exceed a set of memory limit. This also includes file system cache. The original paper was presented at Linux Symposium and can be found at Containers. Challenges with the memory resource controller and its performance. Control groups also provide prioritization. Some groups may get a larger set of CPU or disk IO throughput. Accounting. To measure how much resources certain system use for billing purposes and control, freezing groups or checkpoint and restarting. So effectively, as you can see, this control groups is more like Linux library or a Linux kernel element, but allows you to control and slice and dice the operating system in a way that you believe it's going to be useful. Namespaces. The purpose of each namespace is to wrap particular global system resource in an abstraction that makes it appear to the processes within the namespace that they have their own isolated instance of the global resource. Control groups versus namespace. Control groups limits how much you can use. Namespaces limits how much you can see. So effectively with namespaces, you can have mnt, uts, ipc, net, pid, usr and you can only see that and you cannot see what other people are using. So you can effectively have your own mount with your own namespace, but you cannot see the namespaces the other mounts in different other namespaces. Let's now focus on the mount namespace. This is when you actually use the file system. The mount namespace isolates a set of file system mount points seen by a group of process. Processes in a different mount namespaces can have different view of the file system hierarchy. The mounts can be private or shared, and you can also have your own mount space, or you can even set it. The tasks running in each mount namespace doing mount/ unmount will not affect the file system layout on the other mount namespace. So here, in pictures, we have this mount and this mount, they mount to the same mount namespace, number 1, whereas in this one, it mounts with its own namespace. You can have effectively the user different types of mounts that they can use. Then you have IPC. IPC is inter-process. Inter-process namespace isolates the inter-process communication resources. A process or a group of processes have their own. So you can have your own namespace, you can have this as shared memory, semaphores and message queues. Then you can have the same within these two inter-processes with different namespace, but they don't communicate with each others. Containers use namespaces. So effectively now you can isolate the IPC. You can also isolate the file system as well. You can also isolate the network. So you can have a net namespace, isolate the network related resources. So you can have one namespace here, another and another. In end of the day they go under the global namespace and then they go under the same hardware. So you can still isolate it, but it depends on whether the hardware can support that isolation. So if one user uses, let's say you have 100 gigabit lead and one uses like 99 gigabytes per second, it will may congest the other ones. But from the point of the container itself, it does appear as a separate interface. So effectively you're creating virtual interfaces. You can also have process ID namespaces. Process ID namespace isolate Process ID, implemented as a hierarchy. So Process ID namespace is a hierarchy comprised of "Parent" and "Child". So you can have the parent namespace and you have the child namespaces. In this case, you can have for example, containers where they have their own PIDs, and you have also processes that are on the host operating system. So you can speed up, let's say an Ubuntu container and another Ubuntu container and there you can run their own process, isolate it from the operating system. But you can still run your own Process IDs within the operating system itself, as well. There are many other namespaces, like time-sharing system, UTS. Each process has a separate copy of the host name and NIS domain space so that's isolation. In containers useful for init and config scripts that tailor their actions based on namespace, you can have user namespace. So you can have different types of namespaces that effectively isolates the containers for one with each other.

>> So let us go now and see about Docker. Docker is an open-source project that automates the deployment of applications inside software containers, by providing an additional layer of abstraction and automation of operating system-level virtualization of Linux. In the previous slides, we discussed about how the industries in the 1960s, they used to ship goods and how in the introduction of containers we made advantage of packaging the goods in a container that [inaudible] [NOISE]. In a similar idea we now have different app applications that are packets within a container. Now they can seep in different types of cloud or on-premise hardware deployments. We also discussed about virtual machines versus containers. So let's see, in virtual machine, you have physical hardware, hypervisor, guest operating system, files system, application. Within a Docker, within that container, you have physical hardware, host, Docker engine, union file system, and then different types applications. So what this means is, Docker effectively it's a wrap and around containers. Docker makes it easy to build, modify, publish, search and run containers. So it's more like a container balance with ecosystem. With Docker, a container comprises of both an application and all of its deployments. That means you have the container plus the way you can deploy it. Containers can either be created manually, or if source code containers a Dockerfile, automatically. So dockerfile is more like instructions of how do you create a container and how do you package the software within a container. Subsequent modifications to a baseline Docker images can be committed to a new container using Docker commit function, and then pushed to a central registry. So it's more how do you commit the code upstream to a GitHub repository. You can also commit those Docker commit functions up to specific Docker registry [NOISE]. Containers can be built in a Docker Registry using Docker search. Containers can be pulled from the Docker Registry using Docker pull and can be run, started, stopped. So in this case, we have the Dockerfile and then you have the host operating system, and then we effectively say, "Oh, these are the instructions of how you are going to built that Docker container." So that you build a container. Once you have a container, you can push the container to a registry. So that another user can actually pull the container and drag it, or they can pull a drag multiple different types of containers. Let's see the Docker use cases. Docker provides isolation, portability, and provisioning. So effectively with Docker, you can have all of them together. So you can have the container, you can use different types of scripts like chef, puppet, ansible. These are frameworks for automation of scripts that can build a container. Then you can effectively add in your Docker container details of your applications and libraries together [NOISE]. In a simple where we go to the CLI or Linux, we can just do sudo docker run ubuntu. What we'll do is added bin/echo "Hello World", and what it'll do, it will speed up an Ubuntu container, effectively echo or the command line, "Hello World". That's it. You can also do more things. You can say, I want to run a container with a specific variation of Ubuntu, and then just give the sudo access to the bin/bash. So what happen is, once you run this command, effectively, it speeds up a container and now you're ready to work within the container. So you can do the pwd to see the rest of the folders. You can do ls to see the list of the files and so forth. So effectively here, this is a simple as that. This is the host operating system, you say a lot of container with "Hello World", another container to give you access to that. Once you get access, you can do similar commands as you do it in any Linux operating system. [NOISE]

>> So in the previous lectures, we discussed about Containers, what is a container, what is a control group, and what is the elements that you can build in a container itself. Now let's go to the next phase where it's, how do you orchestrate containers? By Container Orchestration, I mean, how do you orchestrate billions of containers across perhaps millions of each virtual machines? So if you try to find the equivalents of that in the freight example we gave in the freight ships, it's how do you actually start putting down all these processes that they do it differently inside, let's say a specific packaging, packaged in a specific way? For example, how do you put these containers inside the freight ship to ship the goods? So this is what we're trying to do. It's more like the logistics from what we discussed in a good space to how do we do that in the container ecosystem. So all containers share the same resources like CPU, RAM, disk, various kernel things, and so forth, and what we need is we need that fair distribution of goods so everyone gets the fair share. So we need also to prevent denial of service attacks. When we think about prioritization, this is a great saying by George Orwell, "All animals are equal, but some animals are more equal than others." So what we did this we did away, effectively, to figure out priorities across containers, we need to figure out how we optimize the use of resources across containers, and how do we do that across a number of virtual machines? Because when you deploy a service at scale, you will deploy tens, maybe hundreds, maybe thousands of virtual machines and maybe thousands of containers on top of that. So we need container orchestration systems that can facilitate microservices architecture by provisioning containers, balancing the container dependencies, enabling discovery. So we want to identify whether this container, if it's healthy or not. Handle container failure. So if a container goes down, then you bring a new one so it rejoins back the applications, the pool of the containers that serve the application. We want to scale the containers and we want say, now you have the containers, next day, I want 10,000 containers. There are different flavors that can do that, one is the Docker, the other one is Kubernetes, and the other Mesos. Kubernetes is probably the one that is the most famous right now, but the other two are also fairly used across the industry. So you have so many possibilities. So the question is, which are you going to choose? Kubernetes is based on Google's experience of many years working with Linux containers. It has a number of abilities like mount persistent volumes that allows to move containers without music data. What this means is now you can effectively have some block storage. You add a block storage or file system to the container. The container moves down, but still, your block storage is somewhere else. It has a integrated load balancer. So with this, now you can actually move the load across different types of containers. It uses a framework called etcd for service discovery. I totally advise you to go and google that and see what it does. Kubernetes use a different CLI, a different API, a different YAML definitions, and it cannot use Docker CLI. So effectively, Kubernetes has its own way that you can access the container through the command line interface. Kubernetes has the notion of pods. So what is a pod? Pod is a collection of containers that are co-scheduled. Pods are groups of containers that are deployed and scheduled together. A pod will typically include about 1-5 containers which work together as a service. Kubernetes will run other containers to provide logging and monitoring services. Pods are treated as ephemeral in Kubernetes framework. Let's now discuss about services and replication controllers. These are two other elements of Kubernetes. Services are stable endpoints that can be addressed by a name. Services can be connected to pods by using label selectors. For example, this is a database container, this is the Cassandra container, and so forth. The service will automatically round-robin requests between the pods. It also uses what we call replication controllers. Replication controllers are used to instantiate pods. So effectively, if we want to set up a specific set of containers, as we said, a pod, then that's what a replication controller does. Replication controllers control and monitor the number of running pods. They call them replicas for a service. So effectively, we create these replicas of services across the board so that people can access these pods, these set of containers. So it makes it easy to effectively load-balance. Kubernetes has a flat networking space. Containers within a pod share an IP address, but the address space is flat across all pods. All pods can talk to each other without any network address translation. This makes multi-host clusters much more easy to balance at the cost of not supporting links and supporting single host networking a little tricky. As containers in the same pod share an IP, they can communicate by using ports on the local host address. So the beautiful about that is that you don't really need to think about IP addresses translation. Within the pod, you have the same, you have one IP, which means that you can just access each of the containers with pod address. Labels are key value pairs attached to objects in containers, primarily pods, used to describe identifying characteristics of the object, like the version, the tier, the frontend. Labels are not normally unique. They're expected to identify groups of containers. Label selectors can then be used to identify objects or groups of objects. For example, all the pods in the frontend tier with environment set to production. So if you think about that, if you want to deploy a set of pods together, in production, what you do is you create these labels. So if you have 100 pods that you want to deploy or you want to scale to one million pods, all of a sudden, you can use this label to do that. So that makes it easy to effectively balance the number of containers within the pod or even the pods themselves. Another system is called Docker Swarm. Docker Swarm was developed by Docker. Docker is a company,

so they have their own framework. It exposes standard Docker API, meaning that the territory that you use to communicate with Docker, like Docker CLI, Docker-compose, Dokku, Krane. These are all tools within the Docker ecosystem. They can work with Docker Swarm. Of course, they're bound by the limitations of Docker API. So if you think about a Docker Swarm, you can think about having this Docker containers and then you have a Swarm, which is a set of containers coming together. Then you have the CLI, Docker CLI, that you could access also the Swarm as similar way that you are accessing the Docker containers. So in architecture, each host runs a Swarm agent and one host runs a Swarm manager. On small test clusters, this host may also run an agent. The manager is responsive for the orchestration and scheduling of containers on the host. It's obvious. Swarm can be used in a highly available mode where one of etcd, Consul, or Zookeeper is used to handle fail-over to a backup manager. So Consul in Kubernetes, they're different frameworks that you can find online. There are different methods for how host are found and added to a cluster, which is known as discovery in Swarm. By default, token-based discovery is used where the addresses of hosts are kept in a list stored on the Docker Hub.

>> Let us discuss another container management system called Mesos. Apache Mesos is an open-source cluster manager. It is designed to scale to very large clusters involving hundreds of thousands of hosts. Mesos supports diverse workloads from multiple tenants; one user's Docker containers may be running next to other user's Hadoop task. So bottom line, this is basic infeasible with any other container map-making system like Kubernetes and so on and so forth. Apache Mesos was started as a project from University of Berkeley before becoming the underlying infrastructure used to power Twitter and an important tool at many major companies such as eBay and Airbnb. So this is another good example of how like an academic project will become a major industry effect in terms of how we deploy containers. With Mesos, we can run many frameworks simultaneously. Marathon and Chronos are the most well known. I highly suggest you Google those Marathon and Chronos, and read more about those as they correlate to Mesos itself. Let's see now the Mesos architecture. The Mesos architecture is comprised of the Mesos agent nodes, Mesos master, and zookeeper. Mesos agent nodes are responsible for actually running the tasks. All agents submit a list of their available resources to the master. There would be typically 10-100,000 of agent nodes. The Mesos master, the master is responsible for sending tasks to the agents. Maintains a list of available resources and makes "offers" of them to the frameworks. Decides how many resources to offer based on an allocation strategy. There will be typically be two or four stand-by masters ready to take over in case of a failure. So this is for high availability. You can see we have multiple replicas of a master. One probably is in active and the rest are in passive mode. Zookeeper is used to elections and for looking up address of current master. Typically, there are 3-5 Zookeeper instances will be running to ensure availability and handle failures. Bear in mind that in a ZooKeeper, as a centralized configuration management system, I would say as electron system is used by many systems including Mesos. So ZooKeeper is not very much specialized Mesos, but these are widely used database. Frameworks. Frameworks coordinate with the master to schedule tasks onto agent nodes. Frameworks are composed of; executor, scheduler. So executor process which runs on the agents and takes care of running the tasks. Scheduler which registers with the master and selects which resources to use based on offers from the master. There may be multiple frameworks running on Mesos cluster for different kinds of tasks. Users wishing to submit jobs interact with frameworks rather than directly with Mesos. Marathon is designed to start, monitor, and scale long-running applications. Marathon is designed to be flexible about the applications it launches. It can even be used to start other complementary frameworks such as Chronos. Chronos comes from the name "cron", which is used famously as a job executed within Linux. It makes a good choice of framework for running Docker containers, which are directly supported in Marathon. Marathon supports various affinity and constraint rules. That is like different constraints, you can set constraints on CPU, network or memory, or disk drive. Clients interact with Marathon through a REST API. Of course, this is common with many services on the Cloud. Other features include support for health checks and event stream that can be used integrated with load- balancers or for analyzing metrics. Again, we need to support health checks because we need to find out whether the service is up and running, or there is a bug in the system. An event streams means that you can pull information out of Marathon so that you can analyze those metrics.

>> At the final, finalize the discussion about containers with the pros and the cons, and what should you do and what you should not do when they use containers. Containers are immutable. That is, the operating system, library versions, configurations, folders, and applications are all wrapped inside a container. That means that you have a full object or a full entity of what you can write. This guarantees that the same image that was tested in quality assurance would reach the production environment with the same behavior. This is very important, because you can start building early integrations tests for your environment before it gets out the production. Containers are lightweight. That means that the memory that a container takes to be speedup is very small. That keeps the resources utilization at very low amount. So instead of hundreds or thousands of megabytes, the container will only allocate the memory for the main process. Finally, containers are fast. You can start the container as fast as a Linux process takes to start. Instead of minutes, you can start a new container in a few seconds. Instead of minutes, I mean to start virtual machines which takes 50 minutes, 10 minutes, five minutes, and so forth. Container can start in a few seconds. However, many users are still treating container just like typical virtual machines, and forget that containers have one important characteristic, containers are disposable. So when you try to persist data within a container, data may be lost. So let's see now, based on these characteristics of the containers and the properties of themselves, what are the things that one should avoid? One should not persist data in containers. Container just can be stopped, destroyed, and replaced. Which means that, if you start adding data in the container itself, the data will be lost. The only way you can actually write a container with data is to attach a file system to that or attach a block slots device to them. You do not ship the application in pieces. You must ship the application in its totality. One must not create large images. The problem with large images is that it is very hard to distribute to other people. For example, if you're not a yum update which updates all the packages within the Linux operating system, that downloads a new image and becomes large. Do not use a single layer image. To make an effective layered file system, always create your base image for your operating system and then add another layer for the usual definition, then another layer for the runtime installation, and another for the configuration, and another layer for the application. Now, this makes easy to recreate, manage and distribute the image. You can actually include those structures on how you build a container within the Dockerfile. Do not create images from running containers. Like, do not use docker commit to create an image. This method is not reproducible. So which means that you should actually build a container that may commit an already running container. Let's now discuss a little more information about docker. There are different terminologies or how persisted snapshots that can be run. So when you do images, you list all the images. When you do docker run, you create a container from an image and then execute command in it. When you do docker tag, what you do is you tag an image. You do docker pull, is you download that image from the absolute repository. In docker rmi, you're deleting a local image. These are the very basic commands within docker that makes your life much easier. When you run an instance of an image, you do docker ps. What you see is you list all the containers that you have in your operating system. Docker ps minus a, you list all containers including the ones that have been stopped. Docker top is you display processes of a container. Start, you start a stopped container. Stop, you stop a running container. Pause, is you pause all process within the container. Rm is you delete a container. Commit is when you create an image from a container itself. So if we do docker run minus it ubuntu /bin/bash, effectively you run an Ubuntu container. You make some modifications. We can create a new image with the modifications, and then we can run it as a deamon, docker run minus d, and then when the docker logs to figure out whether it's running or not. Dockerfile is docker can build images automatically by reading some sort of instructions from a Dockerfile. Dockerfile is more like a sequence of commands that you include in a file, that docker could use to say, "I'm going to spin up a container with these specific instructions." Dockerfile is a text document that contains all the commands that users could call on the command line to assemble an image. Using docker build, a user can create an automated build that executes several command-line instructions in succession. For example, let's say that you want to install a package in your container. Then what you do is you go and say, install. You say, start a container, install the package, stop the container, and then you can commit the container after. Can be versioned in a version control system like Git or SVN, along with all dependencies. Which means that you can add your Dockerfile with your Git repository in your GitHub account so that when somebody wants to download and to build something, they effectively build it through the instructions of the Dockerfile. There's also Docker Hub, which effectively can automatically build images based on Dockerfiles or GitHub. So Docker Hub is a public repository of Docker images. You can find out many of those. You can find somebody who have published. It's automated, and has been automatically built from Dockerfile. Source for build is available on GitHub. These are very interesting links that I highly advise somebody to read, and these are some of the references that I have also used to build some of these slides. Thank you.

>> So we have discussed about containers, we have discussed about virtual machines, now we're going to the next topic which is serverless. You can effectively run application without even running a single server, this is the most high-end and the most latest topic in terms of Cloud computing that you can probably find around. What is serverless? Serverless architectures refer to applications that significantly depend on third-party services, like Backend as a Service or custom code that runs in ephemeral containers like Function as a Service. So most probably, we'll focus on the second aspect on the Function as a Service, which is how you can run effectively, instead of spinning up a container to run application, you spin up a function that runs in a container So Mobile Backend as a Service, it's serverless, was first used to describe applications that significantly or fully depend on third-party application services in the Cloud to manage server-side logic and state. This is typically rich client applications like single page web apps or mobile apps that use the vast ecosystem of Cloud accessible services, so this could be like Firebase, Parse, authentication like Auth0, or AWS Cognito and so forth. Again, we're not going to focus so much on that, but think about all these elements that tend to run when you run a Mobile as a Service ecosystem. So Functions as a Service applications where some amount of server-side logic is still written by the application developer, but unlike traditional architectures is run in stateless compute containers that are event-triggered and ephemeral and fully managed by a third-party, which means now you don't manage the third-party containers, the containers are ephemeral, and they are triggered by events. So if something happens to the ecosystem, there is an event that gets triggered, spins up a compute container, runs the process and then returns back the outcome, and you don't manage any of the containers, so that's what severless is about. So building and supporting a severless application is not looking after the hardware or the process. Effectively, all of that is outsourced to a vendor, so effectively we're going from Software as a Service, a little higher than Software as a Service I would say, where you don't run anything other than the service itself or the application as a function. Where can you find severless? There are many ways, AWS Lambda is run code without thinking about servers. Pay for only the compute time you consume, which means that you start a function, you deploy it on Amazon, it runs, you get paid, let's say for two, three minutes, five minutes, that this function runs and then it goes down. Google offers a similar [inaudible] , which is Google Cloud Functions. It's a lightweight, event-based, asynchronous compute solution that allows you to create small, single-purpose functions that respond to Cloud events without the need to manage a server or a runtime environment. This of course holds for Google, for Amazon as well, and for Azure Functions and IBM OpenWhisk so that effectively different offerings between Cloud providers are through containers. A good list can be found in this GitHub link, I highly advise you to take a look about that and there you can see effectively all the list of serverless frameworks. The phrase serverless doesn't mean servers are no longer involved, it simply means that the developers no longer have to think that much about them. Computing resources get used as services without having to manage around physical capabilities or limits. Like this, you don't need any more to think about how much memory I'm going to allocate, how much hard drive I'm going to allocate, this is taken care by a third-party provider. Serverless allows you to not think about servers, which means you no longer have to deal with over/under capacity, deployment scripts, Dockerfiles, scaling, fault tolerance, operating system, or language updates, metrics, and logging. All this will be taken care by the Cloud provider, you just write a function.

>> As we go deeper and deeper into the subject of serverless and containers, many of you will wonder. Now we learned a little about container management systems. We did introduction of serverless, where people can now speed up functions to do some work without really needing to understanding the inner pieces of a container ecosystem. How do you really compare those two? Let's take containers as an example. When you deploy a Docker container, let's assume you deploy that on AWS, on Amazon ECS, which is the service that Amazon provides for containers, you still need to think about the hosting cluster that your container will run on. You need to consider questions like: Which cluster will be best placed to run this container? Does this cluster have capacity for my container's resources like CPU and memory? If not, how should I expand the capacity? What is my strategy for deploying multiple instances of the container across multiple machines in the cluster? If the cluster has multiple types of machines within it, do I need to be concerned about when I choose my deployment strategy? What are the security constraints of the cluster, and do they need to be changed in order to properly host may container? So all of these items come to the picture when you start managing containers. You're not managing VMs, but you are going to managing containers. The benefits of serverless is that the hosting provider figures out all the allocate questions for you dynamically. So for example, they will speed up enough containers to run your function. They will do all this balance in the background without the application developer to need to know about the infrastructures, it becomes serverless. It guarantees that it will have enough capacity for your data. If it doesn't, it won't speed up the function. You don't have to spend money upfront over-provisioning your host environment. You do not have to spend money to maintain a container management system. So you're not be constrained down the road by under-provisioning the environment or over-provisioning at the same time. You're not paying for idle time. For example, your function has completed or your application has completed, now it doesn't have to be idle. Reliability, availability are also built it, so you don't have to think about these aspects when you actually speed up functions. By composing and combining different services together in a loose orchestration developers can now build complex systems very quickly, spend most of their time focusing on the business problem than focusing or maintaining infrastructure from VMs to containers. These serverless systems can scale, that they can grow, they can evolve without developers or solution architects, having to worry about remembering to patch the web server yet again. Now, you're focusing on the business logic than focusing on running a web server. For example, what happens if you talked at server which runs the web server goes down? You don't need to think about those items anymore because your service providers provides that. A good serverless architecture can speed up development time and help to produce a more robust product.

>> Serverless containers is deploying functions containers about deploying isolated resources within a machine. Let's now go deeper a little about an example of how serverless, what is serverless? How does serverless work? So let's think about a UI driven application, which is the Classic Design. You have what we call the traditional 3-tier client oriented system with server-side logic. A good example is a typical e-commerce application, for example, implemented in Java on the server side. So the server is actually running some sort of a Tomcat server with an HTML JavaScript component as the client. So effectively, the client asks for the Pet store, I want to buy this food for my dog. The Pet store itself acts as the database, and the database says yes, I have this food, here it is, or get I'm going to run into my car I'm going to buy them. With this architecture, the client can be relatively unintelligent. All right? With much of the logic within the system, like authentication, page navigation, searching, transactions implemented by the server logic. Serverless decide for the Pet store. Now what you need is deleted the application logic, replaced it with a third party service, all right? So in this case we have [NOISE] the client, called access service that says, I want to authenticate a faction speeds up that says, I've got to authenticate that person, then, you check the database, the client itself is the third one, and then you do the search, you type what you did, and then you finally do a purchase function. So for every function that you do within the website as a customer, that speeds a function for the serverless component. So client can have direct access to a subset of our database, for example, just a product listings, which itself is a fully third party hosted. For example, could be like, hey, I'm Amazon AWS DynamoDB has all my data, follow the data, follow the product listings and nobody else can access or other functions. Likely to have a different security profile for the client acts in the database this way from any server instances at the access to the database. So now you're actually isolating the Purchase Database, that Product Database and only specific costumers can have access based on specific privileges that the authentication service has provide. So these previous two points imply a very important that some logic that was at the Pet store server, is now within the client. For example, keeping track of a user's session at the starting at the UX structure of the application like page integration. Reading from a database of translating that into some usable form. The client is in fact well on its way to becoming what we call a single-page application, Server UX related functionality, can be get in the server. For example, if it's computed testable request access to significant amounts of data. An example would be assets. That means that, the reason we do that is because especially with mobile phones, you tend to focus much of the energy conservation, which means that if somebody is computed density has to be pushed the server. For the search feature, instead of having an always running server, we can implement functions as a service function that responds to HTP request via an API Gateway, all right? Which we'll describe later. [inaudible] We have both the client and the set of function read from the same database for product data. Since the original First server was implemented in Java and AWS Lambda, our Faas vendor of choice in this case, supports functions implementing Java, we can port the search code from the Pet store server to the Pet store search function without re-writing anything. Finally, we can replace our purchase function, all right? With another Faas service, to keep it on the server side for security issues rather than implementing the client. It's too fronted by the API Gateway. So what we saw here, is we saw that we can now replace all of these modeling of the web server with much smaller search functions that can run for a finite amount of time. That settle the purpose of that function, like the purchase factor, the search function, and that's bottom line of the idea of serverless

>> So let's go to another example, the Message Driven Serverless. Assume that you have a backend data-processing service, so your writing a user-centric application that needs to quickly respond to UI requests, but secondarily, you want to capture all the different types of activity that are occuring. So let's go to a more concrete example, an online advertisement system. When a user clicks on advertisement, you want to very quickly to redirect them to the target of the advertisement, but at the same time, you need to collect the fact that the click has happened so that you can charge the advertiser. Most famous web platform that you know right now, use are working like that. Whether this is Google or Facebook or other companies database. The revenue advertisements, they click, but they at the same time need to charge the advertiser for that revenue. In the very classical design, the advertisement server synchronously responds to the user. We don't care about that interaction for the sake of this example, but it also post a message to a channel that can be a synchronously processed by a click processor application that updates the derived database to decrement advertisements, the advertiser's budget or that a click has happened so that they can show statistics back through the advertiser. So this is a very well-known example that happens most advertisements websites. So now let's go and see what happened if we switch to advertisement with servers. In this case, competitive provision. The click processor will run as a function and then it will add every time there is a message in the queue. The change in architecture is much smaller here compared to our first example. This is why asynchronous message processing is very popular use case for serverless technologies. We've replaced a long-lived consumer application with a FaaS. This function runs within the event-driven context the vendor provides. Note that the cloud platform vendor supplies both the message broker we discussed and the FaaS environment. So the two systems are effectively close tied to each other. Fundamentally, FaaS is about running a backend code without managing your own service systems or your own long-lived linked server applications. That second clause, the long-lived sever applications, is a key difference when comparing with other modern architectural trends like containers or platform as a service. Now, if we go back to our previous example, the click-processing example, FaaS replaces the click-processing service, possibly a physical machine, but definitely specific application with something that doesn't really need a provisioned server, nor an application that is running all the time. So effectively, FaaS is when what runs at it real quick and added completed and move on. A second property of FaaS is that the FaaS offerings do not require coding to a specific framework or library. FaaS functions are regular applications when it comes to language or environment. For instance, you can speed up an AWS Lambda, whether this implemented JavaScript, Python, Go or any JVM language, or even any. Net language itself. Now the deployment is very different from traditional systems since we have no server applications to run ourselves. In a FaaS environment, we upload the code for our function to the FaaS provider, and the provided does everything else necessary for provisioning resources. That is, whether it is to instantiate a vision of a [inaudible] a process or something else that is really happened so that the FaaS can work properly. The third property is at horizontal scaling is completely automatic. It is elastic and it is managed by this provider. In more other resources you have to balance the scaling of the system. So if, for example, our system needs a 100 requests in parallel the provider will handle that without extra configuration on your part. The "compute containers" executing your functions are ephemeral. That is you should not store any data, with a FaaS provider creating and destroying the purely driven by us, the runtime really needs to do that. Now, most importantly, with FaaS the vendor handles all the resource provisioning and allocation. So there is no cluster VM, there is no Docker management. There is nothing that you need to do that for the container or the virtual machine itself. That's why it is focusing for the serverless idea. Now, if you want to read more resources, these are some great act, is actually some of the code that are within the slides can be retrieved from these resources, as well. There are a lot bodies resources if you want to contact me, I can send you a lot more. Thank you very much.